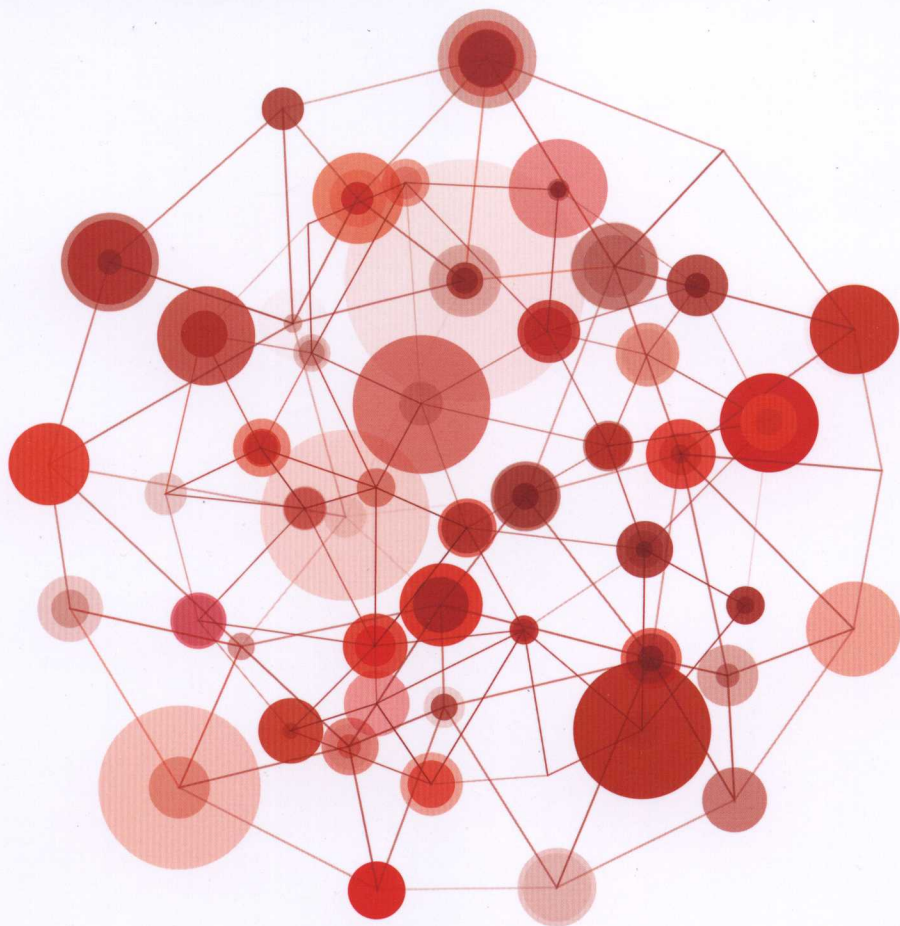


### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF





# IT 基础架构 系统运维实践

赵旻 ◎ 著

站在 IT 基础架构视角，分析数据中心选型与规划、管理流程设计与实施、基础服务构建、系统运维实用经验、职业发展探讨等

资深系统运维专家撰写，知名运维专家联袂推荐，注重方法和思路，将枯燥的操作上升到设计和建模高度



机械工业出版社  
China Machine Press

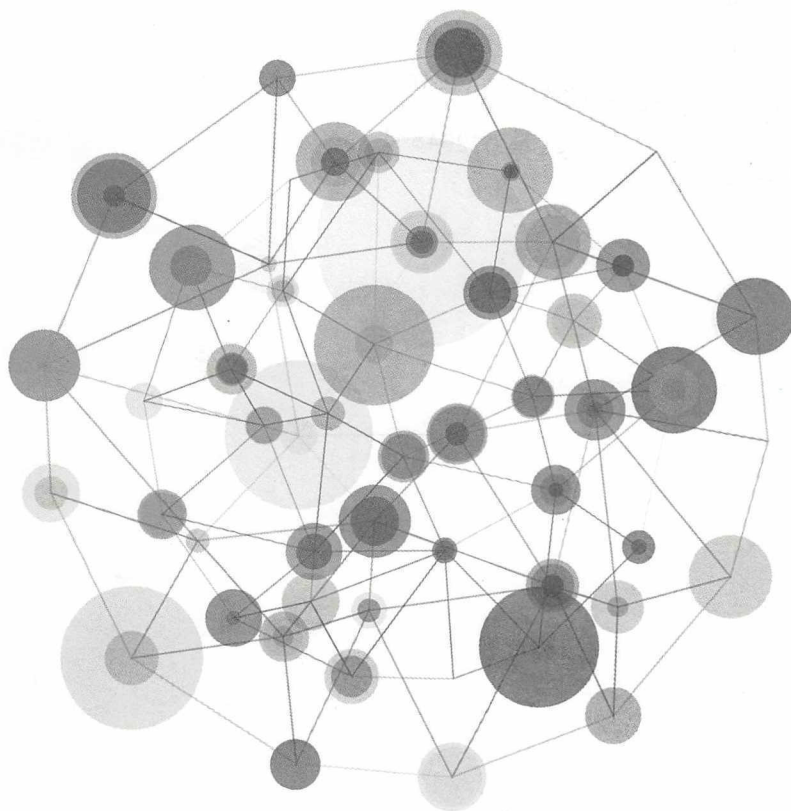
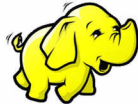
## 作者介绍

---

**赵旻** 获得RHCA/RHCSS/MCITP认证，十年以上互联网金融、电信、政府等多领域背景的从业资历，曾参与中国国家电子政务多项重点工程的安全信任体系建设工作，为中国移动、中国航空等大型企业提供技术支持。熟悉x86平台基础架构系统的建设、管理及运维工作，并醉心于运维产品的设计与体验。乐于在工作实践中分析问题、总结经验，具有持续优化的能力，属于主动管理型的工作者。

资深面试官，产品设计评论人，《运维前线》联合作者，现专注于管理学、产品设计、基础架构运维等领域。

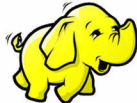




# IT 基础架构 系统运维实践

赵旻 ◎ 著

 机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

IT 基础架构: 系统运维实践 / 赵旻著. —北京: 机械工业出版社, 2018.5

ISBN 978-7-111-59778-0

I. I… II. 赵… III. 计算机系统—研究 IV. TP303

中国版本图书馆 CIP 数据核字 (2018) 第 072948 号

## IT 基础架构: 系统运维实践

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 殷虹

印刷: 北京市兆成印刷有限责任公司

版次: 2018 年 5 月第 1 版第 1 次印刷

开本: 186mm×240mm 1/16

印张: 27.75

书号: ISBN 978-7-111-59778-0

定价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

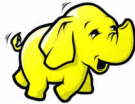
版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东







## 本书赞誉

本书是一本比较难得的好书。基础架构运维工作既非常重要，又很难给出清晰描述。“运维是个筐，啥都往里装”，很多人以为基础架构运维好做，实则不然。

从数据中心水电到网络，从服务器硬件到运维自动化，十八般武艺，既要求知识深度和广度，又事无巨细。

本书以提纲挈领式的全面讲解，呈现了基础运维工作的内容，并将各个要点有机串连起来，深入浅出地贡献给读者，既可以作为基础运维工作的入门书籍，又可以作为日常运维工作的参考比照。

除了享受作者的文笔，不禁还要感谢作者为基础运维工作做出的贡献。推荐！

——刘浩，360 云事业部总经理

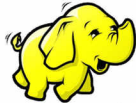
赵兄是一个非常有经验的运维人，有着非常全面而丰富的运维知识。本书作者从数据中心、管理流程、基础服务、系统运维等多个维度来讲解对运维的理解，兼顾了流程、人和技术三个要素，更以层次化的方式自底向上剖析运维，这对一个技术运维人来说是何等的重要。喜欢运维的你，一定不要错过本书的精彩内容！

——王津银，优维科技 CEO

非常高兴看到这样一本系统运维管理领域的专著出版。十多年前刚入行的时候，对系统工程师（SA）这样的职位还很陌生，之后随着支付宝系统规模的暴增，经历了一个难忘的“被成长”经历。同为支付公司运维同行，我和本书作者赵旻的成长经历类似，本书激发了我强烈的共鸣，每一个章节都能让我回想起成长的一段经历。本书绝对是来源于一线实战，又兼具理论高度。云计算时代，对系统工程师的需求和要求越来越高，希望本书的问世可以惠及更多有志于从事云计算运维的同行，传道授业解惑，让天下没有难运维的数据中心！

——智锦，杭州云霁科技 CEO、资深运维从业者

当今互联网正处于快速发展的关键阶段，人工智能、大数据、VR、AR 等新概念的背后，



是基础架构与底层系统支持的发展和实现。没有扎实的基础架构，一切概念与创新都会变成空中楼阁。无论你是运维老鸟，还是刚刚入职的新人，这本书都有适合你的地方。

本书涵盖了数据中心规划、基础服务、系统运维等多个方面。作者以十多年的经验告诉各位读者，弯路一定是会走的，但是如何能够尽早避免，并通过行之有效的方法进行解决，才是运维管理的王道。虽然 IT 界一直在不停地变化，但是运维的核心精神并没有变。本书就是作者多年的运维经验的积累和沉淀，总结出一套颇具心得的 IT 基础架构管理法。

——李晞岩，戴尔（中国）有限公司互联网技术团队经理

本书从机房、电力、服务器等系统管理员的日常工作对象入手，既对企业 IT 基础架构中的各个组件进行了详细阐述，又对一线工作经验进行了提炼和升华。书中通过趣味案例的方式来传达知识，不但增加阅读乐趣，也让读者更容易理解作者要表达的重点内容，同时诱发读者思考故事中的技术点或技巧。IT 基础架构是业务系统的运行基础，稳定安全是首要任务。同时，书中有关系统管理员日常工作的规范和技巧，可以帮助读者解决如何减少沟通成本、如何提高工作效率等问题。

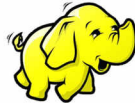
——刘小成，热璞科技咨询交付部总监

本书内容从工作实践出发，是作者多年积累的运维经验和感悟汇聚之作。更加难能可贵的是，作者从通俗易懂的角度，图文并茂地讲述每一个主题，从而让大家理解系统运维管理之道。我认为，不同技术阶段的运维人员都能从书中吸取有益的内容。

本书也可作为运维管理建设中的指导性书籍，或许你能从书中发现自身企业运维管理的痛点，并进行有针对性的改善。感谢赵旻能给运维从业者带来这么好的分享。

——齐代英，贝壳金控资深运维工程师





## 序

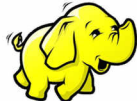
和赵旻认识很长时间了，赵兄技术专业，经验丰富，逻辑性强，善于总结。当前，人工智能、物联网、云计算等新兴技术层出不穷，同时也带来数据爆炸性的增长。为了应对数据爆炸性的增长，保护这些信息的安全，基础设施的可靠、稳定尤为重要。离开了稳固的基础设施，就没有我们的现代化社会。基础设施涉及从硬件到软件、从技术到流程各个层面的知识，内容非常广泛，如何管理好基础设施，知识点非常多。赵旻的这本书很好地、系统化地总结了基础架构知识，相信能够帮助工作中涉及基础架构的工程师，快速成为基础架构领域的运维专家。

本书有三个特点。

**第一，道术结合。**每部分内容，从原理开始，然后介绍实践。因为作者从事基础架构多年，经验丰富，在工作中不断地总结升华，所以原理总结得深刻简明，切中要害。实践方面不但都是干货，而且在组织方面充满逻辑性。阅读本书的时候，读者会觉得非常有条理感，很容易接入自己的知识体系。

**第二，广泛的维度。**本书涉及广泛的基础架构知识，既包括空调水电、硬件、网络、软件方面的技术，又涉及流程、管理等方面的知识，并分享了工程师自我提升的经验。每一部分都不是简单地提及，而是深入论述，许多部分还附有案例。

**第三，充满诚意，充满干货。**本书有 13 个运维故事，每个故事都是作者精心选择或设计的，可见作者为了表述清楚自己的观点，下了很多工夫。许多知识点，作者都进行了详细的讨论。例如关于闰秒这一节，作者讨论了“闰秒是什么，闰秒的危害，前辈们是怎么解决闰秒的，晦涩难懂的术语，怎么解决闰秒问题”几个话题，通过这一节的阅读，读者不仅可以对闰秒有深刻认识，而且还知道了闰秒相关的历史。再如关于服务器硬件的测试，作者分为“测试前的准备工作，部署系统测试，产品功能性测试，能耗测试，CPU 性能测试，内存性能测试，磁盘性能测试，网络性能测试，测试后的收尾工作”几个主题，通过这一节的阅读，读者可以对服务器测试的知识点了解得清清楚楚。类似的地方，书中还有很多，等待着读者来体会发现。

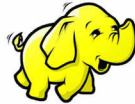


本书是强强联合的出品，本书的作者赵旻是业内著名基础架构专家，本书的编辑高婧雅是知名编辑。高编辑我非常熟悉，工作方式用一个词总结就是“认真”，在选题审稿方面会和作者死磕，正是因为高编辑的认真，高编辑参与的书基本都是精品，本书也不例外。

肖力 云技术社区创始人







## 前言

2015年，国务院政府工作报告中提出制定“互联网+”的行动计划。在这个大背景时代的推动下，越来越多的传统行业面临着与云计算、大数据等热门技术相结合的发展趋势。在漫长的转型过程中，传统企业的IT部门面临着基础架构变革的严峻考验，运维团队不可避免地遇到了很多棘手的难题。例如，管理模式如何由集中式向分布式转型，小型机到x86的演变，海量运维模式的挑战，以及知识结构与运维思路的转变，等等。这些都是目前传统行业IT部门领导者所面临的主要问题。

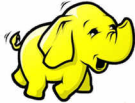
随着电商的流行，也有很多非IT领域的成功企业正在酝酿着自己的O2O市场，希望借助互联网完成第二次创业。他们遇到的最大问题就是——对互联网的认知完全是一片空白。要实现从无到有的原始积累，会有很多挑战在等着他们。

### 为什么要写这本书

基于上述这些问题，我们策划了这套IT基础架构丛书。作为这个系列的第一部作品，我个人的压力还是蛮大的。当机械工业出版社华章公司的高婧雅编辑和我约稿时，自己竟然一时有些不知所措。算起来，我从事系统运维的工作已满十个秋冬。说来惭愧，我觉得自己并没有什么拿得出手的成绩。不论是实践还是基础，市面上这方面的书已经非常多了。那么，以什么作为出发点是合适的呢？最终，我还是从《运维前线》<sup>①</sup>这本书中获得了启发。2017年3月，由云技术社区创始人肖力发起并策划的《运维前线》成功出版，让我感受到了同行们乐于分享的热情，同时也看到了广大读者对实用、落地的技术方案的渴求与肯定。于是，我产生了一个新的想法：在《运维前线》主打实用的基础之上，围绕着我所擅长的系统运维方向，写一部《运维前线》的“系统版”。

---

① 已由机械工业出版社出版，书号978-7-111-55697-8。——编者注



## 本书特色

不管怎么说,技术是一个很枯燥的东西。我自己在学习的过程中也深有体会。拗口的描述、复杂的逻辑是很多技术文档的通病。也许这样的表达形式是严谨的,但它并不“亲民”。我认为,一本好书不但要有深度,更要带领读者一同到达才行。这个深度就像西游记中的水帘洞,如果只有你自己进去了,却把读者晾在一边,那真是太糟糕了。如果一本书洋洋洒洒几十万字,读者看完后没有任何收获,那我宁愿不去写它。因此,打比方和举例子是我在全书中用得最多的写作手法。通俗易懂,是我在技术分享时所秉持的一贯态度。我希望消除掉一切阻碍的门槛,让每一位读者朋友都能够从本书中获得些许的帮助。

选择撰写本书是有着特别的意义的。既然是实践,我们首先要保证技术的实用性。但从定位上讲,它又不同于以往的实践类书籍。书中讲述的所有内容都是笔者正在或者曾经使用过的,并将一些经验和观点融在其中。写这本书,也算是对我多年工作经验的一种总结,了却自己的一桩心愿吧。

## 读者对象

说到这本书的定位,我想它对绝大多数从事系统运维的工作者都是有益的。本书需要一点点 Linux 和网络的基础知识作为铺垫,除此之外再无其他要求。对于工作 3~5 年的朋友们,我知道你们已经厌倦了基本的系统管理,但你们也许有点儿迷茫,不知道下一步该如何进阶。对于那些传统行业面临 IT 基础架构转型的系统运维团队,你们可能在系统管理方面经验丰富,但是对大规模、分布式 x86 平台的系统运维却感到陌生。还有那些刚刚到创业公司的“中生代”技术人,你们可能在工作中会遇到更多新的挑战。我想,选择这本书对你们来说是再适合不过的了。当然,如果你早已是这方面的行家里手,也不妨来读读本书。我的一些经验也许能帮到你,我的一些经历也许能让你感同身受,我的一些观点也许能让你会心一笑,只当是我与你之间的一次未曾谋面的技术交流好了。

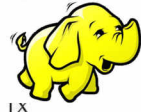
## 如何阅读本书

本书从内容上大致分为六大部分,共计 16 章内容。

**第一部分**(第 1 章),笔者对心中的 IT 基础架构标准、本书的写作初衷和特点等做了阐述。

**第二部分是数据中心篇**(第 2~5 章)。这是一个综合了数据中心、网络、系统等众多技术领域的主题。我作为一个经历过创业公司的老员工,对此深有体会。从无到有,我亲手规划、建设了多个同城的数据中心,后续又和两位牛人学习了很多相关的知识。该篇也许





真的非常跨界，我想在所有讲解系统技术的书籍里，难有雷同之作。作为一名真正的 SE，只懂操作系统是不合格的。所以，我认为这个跨界还是值得的。

第三部分是管理流程篇（第 6 章）。这是一个特殊的篇章，因为它特殊到只有一章。如果能够进一步展开，这个主题其实完全可以独立成书。管理流程是基础架构中最为重要的核心组件。我想没有人会反驳这个观点，除非他所运维的节点数量还不够多。

第四部分是基础服务篇（第 7~11 章）。本篇内容基于多机房和海量节点，介绍了如何去构建 DNS、NTP、文件共享、配置管理等一整套服务的方法。

第五部分是系统运维篇（第 12~15 章）。这部分内容主要和日常运维管理的工作相关。例如，硬件故障处理与维修、安全、性能校准、Shell 程序等。如果要做推荐，我会更倾向于数字证书那一章。因为那是我初入行时的专业方向，和数字证书打了这么多年的交道，写这一篇时也算是一种情怀吧。

第六部分（第 16 章），这部分介绍系统运维工程师应该具备的素养，以及如何提升自己等内容。

此外，这本书中还有 13 个有趣的运维小故事。它们很像登山时的休息点，如果你读累了，可以在这里歇歇脚，喘口气。其实，故事里面也蕴藏着很多收获呢。

不过，这还不是本书全部的内容。既然我受到了《运维前线》的启发，为了表示敬意，我也继承了《运维前线》一书的设计形式。最后一章，藏着一个有趣的彩蛋，等待着读者朋友们去发现。好了，我想我说得已经够多的了，我们在书中相见吧。

## 勘误和支持

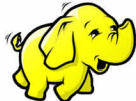
由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如有任何反馈与想法，请你发送电子邮件到 [itarch@qq.com](mailto:itarch@qq.com)。真诚地期待能够得到你的反馈，在技术之路上互勉共进。

## 致谢

在写作这本书时，我得到了很多朋友的帮助。例如我的同事——张望和徐铁军两位大牛。张望是网络方面的专家，铁军则有着多年的 IDC 管理经验。撰写数据中心篇章时，关于一些技术问题的求证，两位给予了我很多的支持与帮助。能和你们在一起工作真好，谢谢两位。

感谢云技术社区的北极熊，熊总在各大社区中不遗余力地帮忙推广本书，做了很多无私的工作。感谢我的那些新老朋友们，在我成书之时，他们帮我撰写书评，给了我很多的鼓励与支持。谢谢你们的帮助与肯定。



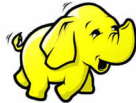


此外，在这里我还要特别感谢两位老师。一位是云技术社区的肖力，另一位是机械工业出版社华章公司的高婧雅老师。两位老师是指引我走上写作道路的领航人，虽然都只有一面之缘，但他们却给我提供了很多的帮助和支持。2015年，我加入了力哥发起的《运维前线》的写作团队。也正是通过这次写作得到了婧雅编辑的肯定，进而才有了这部书稿的成文。力哥在百忙之中亲自为我作序，婧雅为我的写作提供了很多有价值的指导意见。可以说，没有两位就没有这部书的出版。谢谢所有支持我、关心我、帮助我的朋友们，感激之情溢于言表，谢谢大家！

谨以此书献给广大热爱技术的朋友们！

赵旻





## 目 录

### 本书赞誉 序 前言

#### 第 1 章 混沌初开 ..... 1

- 1.1 我眼中的基础架构 ..... 1
- 1.2 写一本怎样的书 ..... 3
  - 1.2.1 英文书的伤痛 ..... 4
  - 1.2.2 有话直说——这就是我的忍道 ..... 4
  - 1.2.3 当行家说人话 ..... 5
- 1.3 本书声明 ..... 6

#### 第 2 章 如何选择优质的数据中心 ..... 7

- 2.1 概述 ..... 7
- 2.2 空间环境评估 ..... 9
  - 2.2.1 地质环境 ..... 9
  - 2.2.2 空间结构 ..... 10
- 2.3 基础设施评估 ..... 13
  - 2.3.1 电气系统 ..... 13
  - 2.3.2 空调系统 ..... 17
  - 2.3.3 消防系统 ..... 21
  - 2.3.4 弱电与综合布线系统 ..... 22
- 2.4 网络建设评估 ..... 23

- 2.5 服务保障评估 ..... 23

- 2.6 本章小结 ..... 24

#### 第 3 章 数据中心的规划设计

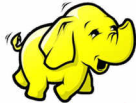
##### 工作 ..... 26

- 3.1 需求的不确定性 ..... 29
- 3.2 如何避免变化打乱规划 ..... 30
  - 3.2.1 采购资源预留 ..... 30
  - 3.2.2 数据中心机柜区域的规划与布局 ..... 31
  - 3.2.3 规划布局案例 ..... 33
- 3.3 规划设计心得 ..... 35
- 3.4 本章小结 ..... 41

#### 第 4 章 网络规划细节对系统运维

##### 的影响 ..... 42

- 4.1 案例复盘 ..... 44
- 4.2 事情为什么弄得一团糟 ..... 48
- 4.3 网络空间资源的规划 ..... 50
  - 4.3.1 PoD 容量的计算方法 ..... 50
  - 4.3.2 地址空间的规划 ..... 51
  - 4.3.3 VLAN 的规划 ..... 52
- 4.4 网卡绑定 ..... 54
  - 4.4.1 网卡绑定模式的选择 ..... 54



|   |                            |           |                                 |                                |            |
|---|----------------------------|-----------|---------------------------------|--------------------------------|------------|
| 4.4.2                                       | 网卡绑定的实现 .....              | 57        | 6.2.3                           | 如何定义你的需求 .....                 | 98         |
| 4.5   | 本章小结 .....                 | 58        | 6.2.4                           | 如何定义表结构 .....                  | 99         |
| <b>第 5 章 服务器硬件选型 .....</b>                  |                            | <b>59</b> | 6.2.5                           | 设计思想原则 .....                   | 103        |
| 5.1   | 如何选择合适的硬件配置 .....          | 59        | 6.3                             | 多面娇娃 Workflow .....            | 106        |
| 5.1.1                                       | 选型的总体原则 .....              | 60        | 6.3.1                           | 一份周报中竟然 80%<br>的工作量都是在沟通 ..... | 106        |
| 5.1.2                                       | 选型中值得注意的地方 .....           | 60        | 6.3.2                           | Workflow 能干什么 .....            | 107        |
| 5.2   | 怎样的一款服务器产品才算是<br>优秀的 ..... | 62        | 6.3.3                           | Workflow 是实例化的<br>规范 .....     | 107        |
| 5.2.1                                       | 带外管理有多重要 .....             | 63        | 6.3.4                           | Workflow 是领航员 .....            | 108        |
| 5.2.2                                       | 异构平台融合能力 .....             | 63        | 6.3.5                           | Workflow 设计中的常见<br>问题 .....    | 109        |
| 5.2.3                                       | 完善的信息数据展示 .....            | 65        | 6.4                             | 本章小结 .....                     | 113        |
| 5.2.4                                       | 软硬件环境兼容性 .....             | 66        | <b>第 7 章 构建 IaaS 平台系统 .....</b> |                                | <b>115</b> |
| 5.2.5                                       | 用户体验 .....                 | 67        | 7.1                             | 高效交付解决方案如何选型 .....             | 117        |
| 5.3   | 产品测试那些事儿 .....             | 69        | 7.2                             | 服务器设置详解 .....                  | 119        |
| 5.3.1                                       | 测试前的准备工作 .....             | 69        | 7.2.1                           | IPMI .....                     | 121        |
| 5.3.2                                       | 部署系统测试 .....               | 70        | 7.2.2                           | racadmin .....                 | 121        |
| 5.3.3                                       | 产品功能性测试 .....              | 70        | 7.2.3                           | SMASH CLP .....                | 123        |
| 5.3.4                                       | 能耗测试 .....                 | 71        | 7.3                             | Cobbler 部署系统详解 .....           | 124        |
| 5.3.5                                       | CPU 性能测试 .....             | 81        | 7.3.1                           | 理解 Cobbler 架构 .....            | 124        |
| 5.3.6                                       | 内存性能测试 .....               | 82        | 7.3.2                           | Cobbler 的安装配置 .....            | 125        |
| 5.3.7                                       | 磁盘性能测试 .....               | 83        | 7.3.3                           | 命名规范 .....                     | 128        |
| 5.3.8                                       | 网络性能测试 .....               | 87        | 7.3.4                           | 创建资源目录 .....                   | 130        |
| 5.3.9                                       | 测试后的收尾工作 .....             | 90        | 7.3.5                           | 创建 Cobbler 部署模板与<br>实例 .....   | 131        |
| 5.4   | 本章小结 .....                 | 91        | 7.3.6                           | Cobbler 里面出现的坑 .....           | 133        |
| <b>第 6 章 构建 CMDB 与 Work-<br/>flow .....</b> |                            | <b>92</b> | 7.4                             | IaaS 系统的设计要点 .....             | 136        |
| 6.1   | 谁拖了运维的后腿 .....             | 93        | 7.4.1                           | 交付工作流程定义 .....                 | 136        |
| 6.2   | 定海神针 CMDB .....            | 94        | 7.4.2                           | Portal 模块与各组件之间<br>的调用关系 ..... | 137        |
| 6.2.1                                       | CMDB 是一切运维的<br>基石 .....    | 95        | 7.5                             | 制作 KVM 虚拟机模板 .....             | 139        |
| 6.2.2                                       | 是什么毁了 CMDB .....           | 97        |                                 |                                |            |

|  |            |                                   |            |
|--|------------|-----------------------------------|------------|
| 7.5.1 虚拟机网络环境部署 .....                  | 140        | 9.1.4 为何要选用硬件时间源<br>服务器 .....     | 187        |
| 7.5.2 创建虚拟机镜像模板 .....                  | 142        | 9.1.5 如何选择硬件时间源<br>服务器 .....      | 188        |
| 7.5.3 虚拟机克隆 .....                      | 143        | 9.2 ntpd .....                    | 191        |
| 7.5.4 虚拟机设备调整 .....                    | 144        | 9.2.1 ntpd 初始化 .....              | 191        |
| 7.5.5 VPC 的支持 .....                    | 145        | 9.2.2 ntpd 配置文件 .....             | 192        |
| 7.6 本章小结 .....                         | 149        | 9.2.3 使用 ntpq 查询时间同步<br>的状态 ..... | 193        |
| <b>第 8 章 构建域名解析服务 .....</b>            | <b>150</b> | 9.3 chronyd .....                 | 197        |
| 8.1 写在前面的话 .....                       | 150        | 9.3.1 chronyd 的优势 .....           | 197        |
| 8.2 首先做好一个传统的 DNS<br>管理员 .....         | 151        | 9.3.2 chronyd 配置文件 .....          | 197        |
| 8.3 Anycast DNS 在多数据中心<br>中的应用 .....   | 171        | 9.3.3 使用 key 限制客户端<br>访问 .....    | 198        |
| 8.3.1 什么是 Anycast .....                | 171        | 9.3.4 跟踪时间同步过程 .....              | 199        |
| 8.3.2 如何构建 DNS over Any-<br>cast ..... | 171        | 9.3.5 检查时间同步状态 .....              | 199        |
| 8.3.3 如何实施 Anycast DNS .....           | 172        | 9.4 如何处理闰秒 .....                  | 200        |
| 8.3.4 如何守护 quagga 进程 .....             | 177        | 9.4.1 闰秒是什么 .....                 | 200        |
| 8.3.5 BGP 在 Anycast 中的<br>应用 .....     | 178        | 9.4.2 闰秒的危害 .....                 | 201        |
| 8.4 HTTP DNS .....                     | 180        | 9.4.3 前辈们是怎么解决<br>闰秒的 .....       | 202        |
| 8.4.1 传统 DNS 的缺陷 .....                 | 180        | 9.4.4 晦涩难懂的术语 .....               | 202        |
| 8.4.2 HTTP DNS 的优势 .....               | 181        | 9.4.5 怎么解决闰秒问题 .....              | 204        |
| 8.4.3 HTTP DNS 长什么样 .....              | 181        | 9.5 本章小结 .....                    | 207        |
| 8.4.4 HTTP DNS 会取代传统<br>的 DNS 吗 .....  | 182        | <b>第 10 章 配置管理 .....</b>          | <b>209</b> |
| 8.5 本章小结 .....                         | 183        | 10.1 本章目的 .....                   | 209        |
| <b>第 9 章 时间同步系统 .....</b>              | <b>184</b> | 10.2 expect 与 Parallel SSH .....  | 210        |
| 9.1 概述 .....                           | 184        | 10.2.1 expect .....               | 210        |
| 9.1.1 如何实现时间同步 .....                   | 184        | 10.2.2 Parallel SSH .....         | 213        |
| 9.1.2 GPS 卫星系统授时原理 .....               | 185        | 10.2.3 SSH 的通病 .....              | 214        |
| 9.1.3 PTP .....                        | 186        | 10.3 Ansible .....                | 218        |
|  |            | 10.3.1 创建 Host Inventory .....    | 218        |



|         |                                     |     |
|---------|-------------------------------------|-----|
| 10.3.2  | 如何自动添加节点                            | 218 |
| 10.3.3  | 组织主机节点                              | 219 |
| 10.3.4  | Ad-Hoc                              | 221 |
| 10.3.5  | Playbook                            | 225 |
| 10.3.6  | 关于优化                                | 231 |
| 10.4    | Puppet                              | 232 |
| 10.4.1  | Puppet 快跑                           | 232 |
| 10.4.2  | 初探 Puppet                           | 234 |
| 10.4.3  | 使用 Apache + Passenger<br>替换 WEBRick | 239 |
| 10.4.4  | Mutil-Master & Mutil-<br>CAServer   | 241 |
| 10.4.5  | 排障                                  | 241 |
| 10.5    | SaltStack                           | 244 |
| 10.5.1  | 配置 Minion                           | 244 |
| 10.5.2  | 管理 Salt Key                         | 244 |
| 10.5.3  | 组织主机节点                              | 245 |
| 10.5.4  | 模块的调用                               | 245 |
| 10.5.5  | Mutil-Masters                       | 247 |
| 10.5.6  | 级联                                  | 248 |
| 10.5.7  | SLS                                 | 249 |
| 10.5.8  | Grain                               | 250 |
| 10.5.9  | Pillar                              | 254 |
| 10.5.10 | 排障                                  | 255 |
| 10.6    | 我们真的能抗住海量节点吗                        | 259 |
| 10.6.1  | 集合编队                                | 260 |
| 10.6.2  | 汇报战况                                | 260 |
| 10.6.3  | 不必过度依赖模块                            | 260 |
| 10.7    | 解决方案的选择                             | 261 |
| 10.8    | 本章小结                                | 265 |

## 第 11 章 文件共享服务 266

|        |              |     |
|--------|--------------|-----|
| 11.1   | 构建 WebDAV 服务 | 266 |
| 11.1.1 | 基本构建         | 266 |

|        |                           |     |
|--------|---------------------------|-----|
| 11.1.2 | WebDAV on HTTPS           | 270 |
| 11.2   | 构建 NFS 服务                 | 272 |
| 11.2.1 | NFS v4 的新特性               | 272 |
| 11.2.2 | NFS 常见问题处理                | 273 |
| 11.2.3 | NFS 高可用方案                 | 277 |
| 11.2.4 | NFS Cluster 实施条件          | 278 |
| 11.2.5 | NFS Cluster 的实施           | 280 |
| 11.2.6 | NFS Cluster 故障排错          | 287 |
| 11.3   | 构建 SFTP 服务                | 288 |
| 11.3.1 | Chroot SFTP 和公钥<br>访问的必要性 | 288 |
| 11.3.2 | 构建 Chroot SFTP            | 289 |
| 11.3.3 | SFTP 容灾方案                 | 294 |
| 11.4   | 本章小结                      | 297 |

## 第 12 章 硬件故障告警与维修 298

|        |                 |     |
|--------|-----------------|-----|
| 12.1   | 硬件故障的特点         | 299 |
| 12.2   | 硬件故障告警          | 300 |
| 12.2.1 | 告警方式            | 300 |
| 12.2.2 | 事件类型和告警级别       | 301 |
| 12.3   | 硬件故障分析          | 302 |
| 12.3.1 | 常用分析手段          | 302 |
| 12.3.2 | 常见故障错误分析        | 306 |
| 12.4   | 传统维修的问题         | 312 |
| 12.5   | 报修系统的需求定义       | 313 |
| 12.5.1 | 故障申报环节的<br>设计需求 | 315 |
| 12.5.2 | 审批通告环节的<br>设计需求 | 316 |
| 12.5.3 | 提交报修环节的<br>设计需求 | 316 |
| 12.5.4 | 设备维修环节的<br>设计需求 | 318 |

|                                      |            |
|--------------------------------------|------------|
| 12.5.5 数据查询统计的<br>设计需求 .....         | 318        |
| 12.6 本章小结 .....                      | 319        |
| <b>第 13 章 主机系统信息安全<br/>基础 .....</b>  | <b>320</b> |
| 13.1 系统安全加固的基本要求 .....               | 320        |
| 13.2 关于安全配置的反思 .....                 | 324        |
| 13.2.1 慎用账户锁定 .....                  | 325        |
| 13.2.2 密码的烦恼 .....                   | 325        |
| 13.2.3 sudo 的意义 .....                | 326        |
| 13.3 sudo over LDAP 的实现 .....        | 327        |
| 13.3.1 服务端配置 .....                   | 327        |
| 13.3.2 客户端配置 .....                   | 329        |
| 13.3.3 关于 LDAP 超时和<br>连接数限制的问题 ..... | 330        |
| 13.4 密码学与数字证书 .....                  | 330        |
| 13.4.1 密码学技术 .....                   | 331        |
| 13.4.2 数据加密与数字签名 .....               | 334        |
| 13.4.3 公钥加密体系的安全性<br>论述 .....        | 336        |
| 13.4.4 数字证书是什么 .....                 | 337        |
| 13.4.5 数字证书是怎么<br>产生的 .....          | 337        |
| 13.4.6 数字证书是怎么<br>验证的 .....          | 338        |
| 13.5 人为因素 .....                      | 340        |
| 13.5.1 运维红线 .....                    | 340        |
| 13.5.2 安全操作 .....                    | 341        |
| 13.5.3 运维工作中的常见<br>问题 .....          | 342        |
| 13.6 本章小结 .....                      | 344        |

|                                   |            |
|-----------------------------------|------------|
| <b>第 14 章 性能校准 .....</b>          | <b>345</b> |
| 14.1 队列理论 .....                   | 346        |
| 14.2 CPU .....                    | 348        |
| 14.2.1 来自内核态的资源<br>消耗 .....       | 348        |
| 14.2.2 用户态资源占用率高 .....            | 353        |
| 14.2.3 Cache 与内存的三种<br>映射关系 ..... | 356        |
| 14.2.4 CPU 调度算法 .....             | 357        |
| 14.2.5 进程运行在哪个<br>核心上 .....       | 359        |
| 14.2.6 strace 的妙用 .....           | 360        |
| 14.3 内存 .....                     | 361        |
| 14.3.1 NUMA .....                 | 362        |
| 14.3.2 Cache 和 Buffer .....       | 364        |
| 14.3.3 虚拟地址空间 .....               | 365        |
| 14.3.4 大页 .....                   | 366        |
| 14.3.5 内存分配 .....                 | 366        |
| 14.3.6 内存回收 .....                 | 368        |
| 14.3.7 内存超配了怎么办 .....             | 369        |
| 14.3.8 为什么会产生 OOM .....           | 370        |
| 14.4 存储 .....                     | 372        |
| 14.4.1 磁盘调度算法 .....               | 372        |
| 14.4.2 I/O 调度算法 .....             | 373        |
| 14.4.3 日志模式 .....                 | 375        |
| 14.4.4 其他因素 .....                 | 376        |
| 14.5 网络 .....                     | 378        |
| 14.5.1 Jumbo Frames .....         | 379        |
| 14.5.2 BDP .....                  | 379        |
| 14.5.3 qperf .....                | 380        |
| 14.5.4 其他 .....                   | 380        |
| 14.6 本章小结 .....                   | 381        |

## 第 15 章 Shell 编程 .....382

## 15.1 参数传递 .....383

## 15.1.1 shift .....383

## 15.1.2 eval .....385

## 15.1.3 getopt .....387

## 15.1.4 函数传参 .....390

## 15.1.5 返回值 .....391

## 15.2 文本处理三剑客 .....393

## 15.2.1 grep .....394

## 15.2.2 sed .....396

## 15.2.3 awk .....397

## 15.3 字符处理 .....401

## 15.3.1 字符的转义 .....401

## 15.3.2 字符串截取 .....403

## 15.4 数组 .....404

## 15.5 算来算去 .....406

## 15.5.1 比较 .....406

## 15.5.2 字符串计算 .....407

## 15.5.3 精度与长度 .....408

## 15.5.4 进制转换 .....408

## 15.6 表面文章 .....409

## 15.7 典型案例 .....410

## 15.8 本章小结 .....416

## 第 16 章 修行之路 .....417

## 16.1 系统工程师的自我修养 .....417

## 16.1.1 工程师与管理员 .....418

## 16.1.2 系统工程师的三颗心 .....419

## 16.1.3 匠人精神 .....420

## 16.2 未来时代 .....422

16.2.1 前方高能——出现怪兽  
AlphaGo .....42216.2.2 从现在开始就要改变  
自己 .....424

## 16.2.3 开启你的管理模式 .....425

## 16.3 写在最后的话 .....427

# 混沌初开

混沌未分天地乱，茫茫渺渺无人见。

自从盘古破鸿蒙，开辟从兹清浊辨。

在这里，首先要感谢你能选择这本书。我已经很多年没有动过笔了。自打高考结束之后，我就很少再有机会爬格子了。这一切还是托了肖力兄的福。通过《运维前线》这部书，我结识了很多同行好友。也承蒙力哥和机械工业出版社华章分社的高婧雅编辑的鼓励与错爱，再一次燃起了我的写作热情。

### 1.1 我眼中的基础架构

相传战国时期，魏国的国君魏文侯外出巡游，在路上遇到了一个反穿裘皮大衣的人。魏文侯觉得很奇怪，向那人问道：“你为何要将衣服的毛穿在里头，却把皮板露在外面呢？”那人回答说：“这件裘皮大衣很贵，我怕损伤了毛，所以才这样穿的。”魏文侯大笑：“难道你不知道，要是这皮板磨破了，毛也就保不住了吗？”

古人云：“兵马未动，粮草先行。”没有必要质疑基础架构的重要性。2015年至2017年可以称得上是互联网界的“三年灾害时期”。重大事故频发，很多知名企业先后“中枪”，在业内引发了很大反响。其中大多数故障与基础架构有关，从这些事件上折射出了很多问题——我们对基础架构的重视和投入是不足的。

其实“平日不烧香”倒也没什么关系，只要事后你能平心静气地承受所有的损失，不要哭天抢地、痛心疾首就好。这和买保险是一样的，投100万元还是100元，完全由你决定。但有一点我能肯定，你觉得你的基础架构值多少钱，你就会去投入多少。

做基础架构，我认为有三点非常重要——可靠、简单和高效。

#### 1. 可靠

可靠是毫无疑问的。如果底层不扎实，上面再花哨也没有用。不少企业在建设初期，



为了抢占市场、追赶进度，相对于产品开发，基础设施的建设往往滞后得非常厉害。我在本书的第6章，特别强调了基础架构中的两大基石——CMDB 和 Workflow 的重要性。试想如果没有 CMDB，你的信息从哪里来？没有规范的 Workflow，如何确保信息的准确性？把精力放在基础架构的建设上，要比盲目求高、求快有意义得多。

可靠性不仅仅体现在软件架构上，基础设施的可靠性同样重要。基础设施在可靠性上栽跟头的主要原因就是钱。财务、采购和技术部门经常在成本价格上发生冲突。一边要节约成本，一边要提升服务等级，每个部门都在追求自己的价值体现。部门的话语权就决定了你是否会掉进低价陷阱之中。廉价、免费是世界上最昂贵的商品。看似便宜的东西，买回来就会大呼上当。不是品质打折扣，就是后期产生无尽的费用增项。这种“抠小钱、舍大钱”的做法是非常不值的。

## 2. 简单

简单是支撑规模增长、消除扩容瓶颈的基础要件。架构的复杂度和今后发生事故的严重程度是成正比的。底层架构的逻辑越复杂，出问题以后牵扯的东西就越多。真是因果循环、报应不爽啊。

不同的体量，架构的实现模式会有不同。因而随着规模的增长，你的架构会不自觉地增加复杂度。这是不可避免的，所以架构师要不断地为新架构减负和解耦。在削减赘肉的过程中，我们要消灭基础架构身上的三座大山。

**第一，要消灭异构形式。**不论是通信协议、接口规范抑或实现方式，最忌讳的事情就是大家各玩各的。你做一个标准，我做一个标准，每接入一个新的子系统，就要产生很多兼容性问题，最后还不是要花大力气去融合么？

**第二，要消灭重复组件。**底层功能的实现在业界大多有成熟的解决方案，应当尽可能地去复用或者在此基础上进行改进，而不是重复造轮子。研发精力要更多地放在业务场景的适用性上。

**第三，要消灭紧耦合关系。**解耦工作要从不同的角度入手，子系统和模块的紧耦合因素是不同的。

实现系统间的解耦要让架构扁平化，消灭冗余的层级。垂直层面上的内容越多，就越容易出问题。如果执行一项任务要历经十个系统。一旦任务失败，这十个系统都脱不了干系。另外，在需求迭代的过程中，每增加一个需求项，不论你所负责的系统是否需要修改，在任务没有明确之前，这十个系统的开发人员都得参与讨论。不断地开会，不断地沟通，你想这些成本代价得有多大？

实现模块间的解耦要明确职权界限。调度模块只负责调度，部署模块只负责部署。假使调度模块在调度过程中发现，环境的初始化工作没做好（例如缺少某个配置文件），那么它应当给部署模块返回错误消息，而不能让调度模块去做拉取文件的动作。如果开发任务指派了多个人去完成，则这种问题就很容易发生。等于两个模块里面都有部署的动作，它

们之间的界限就模糊了，从而出现多头管理的问题。

解耦的核心是拆除过度的依赖关系，让暴露给外部的细节最小化，形成“高内聚、低耦合”的最优状态。

### 3. 高效

团队工作的特点是人多省力、人少省心。当一件工作从一个人干变成十个人干时，你会发现信息来源的权威性、信息同步的实时性和操作指导的规范性有多重要。当运维团队增长到一定规模的时候，效率反而会有所下降。因为工作被进一步拆分了，在缓解压力的同时，相互之间的衔接环节却增多了，沟通成本以及利益冲突的负效应也会在这个时期显著放大。

影响效率的主要因素来自于 CMDB 和 Workflow 模型的成熟度。具体细节我们就不在这里过多讨论了。总而言之，基础架构做不好，整个团队的管理是一片混乱，执行效率低下，成员怨声载道。俗话说：磨刀不误砍柴工。把精力多放在基础架构的建设上是绝对不会吃亏的。

## 1.2 写一本怎样的书

在策划这部书的时候，我受到了《运维前线》的很多启发。2015 年，我有幸加入了首部《运维前线》的写作团队。从《运维前线》的创作到出版，不仅让我感受到了同行们乐于分享的热情，同时也看到了广大读者对于实用、落地的技术方案的渴求与肯定。于是我有了一个新的想法：能否在《运维前线》主打实用的基础之上，再让它更加系统、丰富一些呢？

选择撰写这部《IT 基础架构：系统运维实践》是有着特别的意义的。在定位上，它不同于以往的实践类书籍。当然，既然是“系统运维实践”，我们首先要保证技术的实用性，必须回归到具体操作的实例讲解中来。什么是实践？我想绝对不是在虚拟机上呈现一个实验结果那么简单。如果只是这样，你完全可以在互联网上找到很多类似的文档，这不是我想要的。真正的实践，一定要是你亲身用过的东西，有宝贵的经验之谈在里面。我们从来就不缺少安装文档，读者需要的是——站在架构层面上的技术应用指南。这，才是我写作的初衷。

当然，基础架构这个题目很大，绝非我一人之力所能承担，所以我并非一个人在战斗。我希望能沿用《运维前线》的众筹创作模式，汇聚业内众多专家，总结他们多年的工作经验，从而形成一套通用的解决方案模型，使之可以应用于大部分的场景。也就是说，作为抛砖引玉之作，本书只是“IT 基础架构”系列丛书的第一部。我们正在邀请更多的作者加入到这个项目当中。我相信这是善举，愿这套“IT 基础架构”丛书能成为真正落地的解决方案。就像马云所说的，让技术为大众去服务，而不是成为牟取暴利和树立壁垒的手段。

### 1.2.1 英文书的伤痛

有很多人曾经都问过我，学习 Linux 应该看些什么书。我的建议是，最好先去读 Redhat OD（官方文档）。它的权威性是不言而喻的。阅读 OD 时，你的心里是踏实的。尤其是到达一定技术深度的时候，知识来源的可靠性非常重要。不过很多朋友表示看 OD 有难度，主要是英文不过关，这一点我也是感同身受。

翻译专业文档的确是一件极其辛苦的工作，要做到“信、达、雅”谈何容易？一方面我们对于专业词汇的理解不到位。另一方面，外国人的某些表达方式与我们不同。明明很简单的道理，偏要花几大段英文来解释。而拆解成两三个短句能说清楚的事情，却非得用一个超长从句来表达，还有各种 which 与 it 的不明指代。这让我深感英文依旧是我们技术进步的最大障碍。

在讲解一些来源于英文文献的内容时，为了弄清楚各种专业词汇，我翻阅了大量的资料。例如，在时间同步这一章中，像 slew、smear 这类单词，即便你知道它们的原意，也未必就能很好地将其代入实际语境当中。而且就算翻译到位了，读者对专业词汇的理解也是吃力的。在充分理解原句含义的同时，再用浅显易懂的语言描述出来，我在这方面也着实是费了一番工夫的。

### 1.2.2 有话直说——这就是我的忍道

我中学时代的一位语文老师在我们讲写作时，曾举过一个特别生动的例子。当时中学作文的字数限制是八百字。她说，文章开篇一定要平铺直叙，快速切入主题，字数不要超过一百。写作文就像打仗，这八百个字就是打仗时你率领的士兵。有些同学写作文开篇太啰唆，一冲锋都死伤一百多个兵了，结果还不知道你要说的是啥呢。

事实上，我自己也特别痛恨这种说话兜圈子的方式。如果一部书洋洋洒洒几十万字，读者看完后没有任何收获，那我宁愿不去写它。我不能变成自己所讨厌的那种人。现在大家提倡讲干货，就是因为有些技术分享已经开始变味儿了。举几个例子，我想大家一定都有同感。

#### （1）白皮书式的分享

白皮书式的分享就像是一个销售在推销，全程围绕着一张架构图，依次介绍里面每一个模块的名字和作用，最后展示一张效果图。分享结束后，听众发现自己和这个产品完全没有任何交集。其实，大家并不关心这些模块叫什么，做什么用。我们真正感兴趣的是，整个系统的工作流程是怎样实现的，模块之间是如何协同工作的，在实现过程中遇到了什么问题，又是如何解决的，这套系统适用于哪些场景，在什么时候会遇到瓶颈，这个模式我们能复用么？然而，这些问题都没有给出答案。



## （2）炫富型的分享

我觉得这种分享根本就是来“拉仇恨”的。巴拉巴拉讲了一个特别牛的产品，最后话锋一转，说这是我们团队自主研发的。言外之意就是——你们用不了！

## （3）表演型的分享

技术分享的目的是让大家获益，而不是用来突显你有多高明。有些分享特别热衷于过度包装，非要把简单的东西往复杂里说，刻意地堆砌各种术语以及模型公式，却又不做任何解释。这种做法完全不顾群众的接受能力，似乎一开始就没打算让你弄明白。

从上述这几个问题来看，我认为分享人的内心并不是真正开放的。写书也是一样的道理，读者并不是傻子，你自己内心的拧巴与纠结，大家看得很清楚。

如果非要下一个定义，我认为《IT 基础架构：系统运维实践》是一部抽象化的实践类手册。这里面既有模型和方法，也有案例和步骤。作为抽象化的架构设计，我会交代清楚它的原理、构想以及我们为什么要这样做。作为具体的事例，我要确保它是切实可行的。读者完全可以复用到大多数场景中去，因为它们都是笔者曾经或者正在使用的技术方案。

一部书有没有价值，并不在于技术的深浅，而在于作者的诚意——有没有分享干货的意愿。我愿意尽我所能去帮助任何有需要的朋友，相信这是一部脚踏实地、有一说一的书。有话直说，这就是我的忍道。

### 1.2.3 当行家说人话

我入行是从信息安全做起的，那时我在政府行业中做了很多国家级项目。其中某部委的一位领导，给我留下了很深的印象。有一次，我们的项目经理提交了一份文档，被她给打回来了。她让项目经理当场高声朗读这份文档。这一读不要紧，项目经理自己也觉出来了，文中语病甚多。这位领导斥责道：如果你写的文档连你自己都读不明白，怎么好意思拿给别人去看？

这件事对我的影响很深。程序员写完代码要自测，作者写完文章要自读。而且一定要念出声儿来，这一点很重要。朗读能够从心态上清空自我意识，从一个读者的角度去读自己的文章，你是不是真的能读懂你所说的话呢？

当行家、说人话，是我这次秉承的写作态度。为了避免出现“看不懂、读不通、想不明、理不清”的情况，这一部长达 30 万字的书，我至少写了 50 万字的内容。即便是这样的一个开篇，我反反复复地也删改了很多次。有时我真的很痛苦，利用通宵及周末加班写出的大段文字，最后很多都被自己删掉了。忙了几天，却发现 Word 的字数统计几乎没有变化。我不是职业作家，交稿的时间也很仓促，要说没有压力是骗人的。其实在临近截稿前，我感觉自己就像是一个高考结束铃响时还在奋笔疾书的考生，很想能有更多的时间来细细打磨全书。因为我希望这部书的内容能更加完善，有些呆板干涩的地方，也许还可以将它润色得更好一些。此外，我也担心因为自己的一时疏忽或者知识上的空白，而给本书



留下一些错误。读者朋友们，如果你在阅读的过程中发现了任何问题或疑问，欢迎你提出批评和建议，我将在下一版中进行改进。如有任何反馈与想法，请发送电子邮件到 [itarch@qq.com](mailto:itarch@qq.com)。再次向你表示感谢。

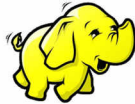
### 1.3 本书声明

在拟定本书目录之时，高婧雅编辑曾经给我提过一些建议，她希望本书能够有一些实际案例的讲解。我原本就有培训的工作经验，而举例子和打比方又是我喜欢的讲述风格，所以列举案例对我来说不是问题。但我又有点儿为难，案例中难免会有些反面典型。但做运维的人，谁没开过故障分析会？谁没犯过错误？案例分析不是要针对谁，而是为了达到总结经验、吸取教训、警醒后人的目的。在此我郑重声明，本书列举案例所涉及的公司和人名均为虚构，一切只为说明问题，请勿对号入座哦。你要是较真儿，那我就只能说清者自清了。还是那句老话，本书故事纯属虚构，如有雷同，实属巧合。

在撰写本书时，笔者参考了很多非常有价值的资料。有些源于 Red Hat 的官方文档或软件的用户手册，一些名词解释和介绍性的内容来自互联网，还有一部分摘录自笔者已经发表在《运维前线》上的文章。

除此之外，在撰写本书的第 8、10 和 14 章时，笔者还参考并借鉴了如下书籍中的部分内容。在此，我对这些书籍的作者们深表敬意。

- [1] Cricket Liu, Paul Albitz. DNS and BIND [M]. 5 版. 纽约: O'Reilly Media, 2009.
- [2] 克鲁姆, 等. 精通 Puppet 配置管理工具 [M]. 李超, 译. 2 版. 北京: 人民邮电出版社, 2014.
- [3] 李松涛, 魏巍, 甘捷. Ansible 权威指南 [M]. 北京: 机械工业出版社, 2016.
- [4] Joseph Hall. 精通 SaltStack [M]. 姚炫伟, 译. 北京: 电子工业出版社, 2016.
- [5] 莫尔勒. 深入 Linux 内核架构 [M]. 郭旭, 译. 北京: 人民邮电出版社, 2016.
- [6] Redhat, Inc. RedHat. System Monitoring and Performance Tuning: RH442 [EB/OL]. <https://www.redhat.com/en/services/training/rh442-red-hat-enterprise-performance-tuning>.



## 第 2 章

# 如何选择优质的数据中心

信息化是当今世界经济和社会发展的趋势。谁能够掌握更多、更准确、更及时的信息数据，谁就能在这个时代的发展大潮中占据主动与先机。毫不夸张地讲，对信息数据的有效掌控，将成为企业的核心竞争力。

对于小微企业和个人创业者而言，租用公有云是个不错的选择。尤其是那些非 IT 企业，租用公有云可以免除日常维护的烦恼。然而，随着企业规模的不断壮大与数据量的急剧增长，公有云的弊端就会逐步显露出来。

第一，业务增长导致存储和计算节点的需求激增，成本价格会成为扩容的最大障碍。

第二，公有云数据托管的安全问题一直备受争议。作为命脉的核心数据，又岂能轻易地托付于人？

因此，公有云的应用是有很大局限性的。

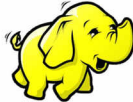
在“互联网+”这个大趋势之下，IT 基础架构也面临着变革的考验，以 x86 平台为代表的大规模集群、分布式的架构必将成为主流。因此，建设一个高可靠、大容量的数据中心的必要性就不言而喻了。

## 2.1 概述

数据中心的建设有两种方式。

第一种是采取机房自建的方式，它的优势在于满足定制化需求和后期的成本控制。但是，对于大多数企业来说，机房自建是一个相当复杂而庞大的工程。光国家标准规范就有上百部之多，需要很多专业人员参与，而且初期投资巨大，没有一定的规模，自建是得不偿失的。

还有另外一种切实可行的方式：选择一家优质可靠的数据中心，把自己的设备托管给服务商，这种方案是大多数企业的首选。不过，我们在数据中心的选择上可能没有做足功课，因为选择了一家不靠谱的运营服务商，导致业务中断、数据丢失的案例也不止一两起。



对于数据中心存在认识上的误区，我想这是问题的主要原因。大家可能总是觉得，机房工作不就是上架、布线嘛，很简单啊。那不过是日常工作而已。数据中心的建设和维护其实是一门涉猎非常广泛的学问，涵盖了地理学、建筑学、工程预算、管理学等诸多内容。想成为这个行业的佼佼者，只有技术是不够的，至少要在以下三个方面有所成就。

- ❑ 专业知识——专业知识过硬是最基本的要求。
- ❑ 行业情报——如果一个人深谙行业内的来龙去脉，而且总是能够第一时间了解到最新的相关消息，那么他的价值远非单纯的技术专家可比。
- ❑ 人脉关系——关系的重要性自不必解释。深厚的人脉关系往往能趟平很多看似困难重重的坎儿。

---

### 运维故事 1：水火相容

Forever 和 Never 是两家互为竞争对手的公司。Forever 公司 IT 部的架构师老 Z 跳槽去了 Never，正赶上 Never 正在筹划新的三地多机房数据中心，为了一显身手，老 Z 把 Forever 已经完全成熟的网络架构方案直接照搬给了自己的老板。这个架构的骨干网涉及一个问题：需要两家大型电信供应商 Conflagration（火灾）和 Flood（洪水）之间的链路完成互通。光看名字，就能明白这两家是世仇，可见想要实现竞争对手之间的链路互通谈何容易？老板甚至都有点儿怀疑当初聘请老 Z 这件事是不是有点儿不太靠谱儿。老 Z 自己觉得很委屈，以前在 Forever 的时候明明就是很容易实现的啊？他百思不得其解，于是只能给原来的老同事——Forever 公司负责数据中心的经理老 D 打了通电话。

在听完老 D 洋洋得意的一番解释之后，他这才明白其中的缘由。原来老 D 可不只是个搞技术的，人家在 IDC 行业摸爬滚打了 20 多年，除了技术水平高，更重要的是对这里的市场行情和人脉关系门儿清。Conflagration 的老总和老 D 早先是老同学，俩人私人关系一直不错，而 Flood 的高层和老 D 是多年的好哥们儿。所以，这趟活儿人家是托了人情的，能顺利促成也无非都是冲着老 D 的面子，而非像老 Z 那种想当然。当然，按照老 D 的话讲，他为了这个项目，也是把这些年积攒的人情面子一次性地全给用光了，要不是老板再三保证能给自己升职加薪，他才不会这般自讨苦吃呢。

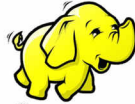
---

既然数据中心如此重要，那么如何选择一家优质的数据中心呢？大家也许会说，那还不简单，找一家口碑好的不就行了？要不，就招一个数据中心专家来解决。我们是做系统运维的，这和我们又有什么关系呢？

其实，在开始写作之前，我还和高婧雅编辑就这个问题多次探讨过。系统运维扯上数据中心会不会偏离了主题呢？会不会让读者觉得枯燥呢？但是当我真正地完成了这章内容的写作时，我已经对此有了足够的信心。

从数据中心的角​​度讲，藤上的甜葡萄早就被人摘光了。好的数据中心有可能因为用户





众多,在容量上无法满足你的需求。另外,数据中心也不是简单看看评级和口碑就能够下结论的,同级别的数据中心之间也会有很多差异。

从人的角度讲,请一位资深专家在成本预算上是比较高的,大多数情况下,这项工作可能是以兼职的形式出现的。而数据中心又是一门广泛而复杂的学问,真要去系统地学习,恐怕短时间内难以有所作为。

实话实说,我并不是数据中心方面的专家,连专业都谈不上,所以我不打算把文章写得过于理论化。相反,我会用一些浅显易懂、轻松愉快的方式来讲述。

## 2.2 空间环境评估

从本质上讲,数据中心也是一栋建筑物。就和我们买房一样,在地理位置和房屋结构的挑选上,也需要仔细斟酌。数据中心的选址工作非常重要,地理位置决定了数据中心的安全性以及工作的便利性,而牢固的建筑结构、合理的空间布局同样非常关键。

2015年8月12日,天津港发生了特大爆炸事件。爆炸同时引发了周边地区多起火灾及二次爆炸事故,方圆数公里都有强烈震感。而位于事发地点附近的某个大型数据中心在此次事故中受到了严重冲击。由此可见,数据中心的选址工作是多么重要。

我们再来看一个场景。某客户近日需要紧急上架200台服务器。但不巧的是,当日天气突变,运输的货车刚刚到达,转眼间就下起了倾盆大雨。数据中心既没有可以避雨的场地,室内也没有充足的空间来临时放置设备,大家只好在心里默默祈祷,希望这场大雨能早一点儿停下来。从这个案例中可以看出场地条件对工作开展的限制与影响。

### 2.2.1 地质环境

我们在选择数据中心的时候,尤其是用于灾备的异地机房,一定要做好充分的考察工作,确认数据中心所处的地理位置是安全的。

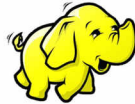
由于我国位于亚欧、太平洋、印度洋三大板块的交界处,这样的地理位置使得构造断裂活动强烈,常年均有不同程度的地质灾害发生。南部地区降水多,泥石流、洪涝灾害比较严重。尤其是西南方向,青藏高原、云贵高原以及四川盆地是地震、山体滑坡、泥石流的多发地带,四川汶川和青海玉树近年来都发生过特大地震灾害。北方的华北地区也属于地震多发带。此外,华北、西北地区还会因为地表水不足和开采过度等原因出现沉降带。而东南沿海方向,主要是受到塌陷和海啸的侵袭。

简而言之,我们选址的主要目的就是为了躲避天灾和人祸。我们要做到五个远离、三个临近以及一个确保。

#### (1) 五个远离

五个远离是指数据中心应远离水、土、火、污、磁这五大区域。





- ❑ 洪涝区域，水库堤坝下游，沿海航道，江河湖泊。
- ❑ 山地滑坡、泥石流地带，火山、地震、地质断裂带，低洼、沉降带。
- ❑ 化工厂、加油站、面粉厂等地。
- ❑ 核电站或者重污染源（包括噪声污染、化学污染、粉尘污染等）。
- ❑ 发电站、变电站、铁路、机场等地会产生较强的电磁干扰，对设备的正常通信和信号接收造成影响。

### （2）三个临近

三个临近是指临近交通、运输和生活区。

- ❑ 临近市政主干道，便于设备运输。应当具有两条以上可直连主干道的道路，确保不会发生运输线中断事件。
- ❑ 临近公共交通站，为来往提供交通便利。
- ❑ 临近城市及生活商业区，方便工作人员在附近就餐及安排住宿，为驻场值守工作提供便利。

### （3）一个确保

一个确保是说，要确保该地区有良好的移动信号，保证联络通畅。

更多的细节，我们可以参考《计算机场地通用规范》（GB/T 2887—2011）。不过，在这里提示大家注意，千万不要走极端。比如加油站，它像一个巨大的储油罐，起火爆炸后产生的冲击波会波及周边的建筑群，应采取远离避让的策略。但加油站同时也是数据中心停电时有力的后援保障。尤其是大面积停电且短时间内无法恢复时，备用的柴油发电机要有足够的油料供应。数据中心可以和加油站签订协议，在特殊情况下保证油料的及时供应。所以在选址的过程中要综合考虑问题，不应按图索骥，过于教条。

## 2.2.2 空间结构

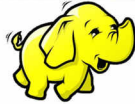
### 1. 机房的抗震等级和载荷

根据《建筑工程抗震设防分类标准》（GB 50223—2008）中的描述我们可以得知，按照建筑物的重要程度以及发生灾害后所产生的影响，可以将抗震等级分为如下四类。

- ❑ 特殊设防类（简称甲类）：使用特殊设施、涉及国家重大工程或者地震发生后会引起严重次生灾害的重点建筑。
- ❑ 重点设防类（简称乙类）：指地震发生时使用功能不能发生中断或需尽快恢复的，以及地震时有可能导致大量人员伤亡的关键建筑。
- ❑ 标准设防类（简称丙类）：一般建筑类。
- ❑ 适度设防类（简称丁类）：适度降低要求的建筑。

对于机房的抗震等级选择，我们建议重点地区的核心机房应当优先选用乙类以上标准，抗震烈度不低于 7 级。机房区域的楼层活荷载应不小于  $1000\text{kN/m}^2$ ，UPS、柴油发电机等安





装区域应不小于  $1600\text{kN/m}^2$ 。另外，室内基础设施与服务器机柜应当采取相应的加固及减震等防护措施。

## 2. 外场地空间的选择

外场地空间的条件是否良好，对设备安装进度会产生很大的影响。互联网公司每年的设备采购量级都很大。一次性运送、拆卸几百乃至上千台设备，对于我们来说太平常了。场地空间大小、货梯的容量与承重、人员和小推车的数量，这些条件决定了你的工作效率。如果场地空间狭小，没有足够的地方卸货，你送货的车辆可能都不一定能开进来。又或者你一次到了 1000 台设备，结果发现只有一部货梯和两辆小推车，每次最多只能运送 30 台设备。你还得考虑资源争用的问题，如果其他用户也有同样的需求，想要按时完工就很困难。再者，工作效率低对人也是一种折磨。参与过上架工作的读者应该非常清楚其中的辛苦，这是个纯体力的工作。严寒酷暑不说，还有雾霾，长时间露天工作对身体的伤害很大。此外，像夏天雷雨多发季，天气多变，如果在卸货过程中突然赶上降雨则更加麻烦。

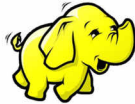
一般来说，送货之前都要提前看天气预报，尽量避开可能降雨的日子。但如果不巧遇到突发天气，大空间的独立库房也是非常重要的，至少它可以保障设备有临时存放的空间。库房的面积和采购运输的设备数量有关，如果用户的保有量很大，在数据中心不同的楼层拥有多个机房，既可以选择分别租用各层库房的部分空间以方便物品的存放和取用，也可以选择租用独立的库房。笔者更倾向于选择后者，我宁愿要一个独立库房，也不想每层都安排一处和别人共用。一来库房过多管理起来不方便；二来租用一个大库房，空间面积的利用率更高，而且设备和别人放在一起也不太安全。至于便利性，相比起来我认为是可以舍弃的。库房主要用于临时存放物品（像存储静置或者临时避雨等情况），设备到货后都应当尽快完成上架工作。所以除了备件之外，库房日常的利用率应当比较低才对。而且我也不建议采购太多备件，硬件设备更新换代快，而 x86 分布式架构也不应该再出现单点服务的问题，何况库房环境不一定符合备件的存放条件。上述诸多因素表明采购备件的意义不大，除了核心设备以外，完全可以在维修的时候再进行更换。

如果你的系统对时间的精确度要求比较高（例如金融行业），需要采购 NTP 时间服务器，那么请不要忽略对建筑物天台的考察。如果天台的条件无法满足卫星天线的安装，则谨慎考虑将其设置为你的主机房。

## 3. 机房的选择

选用机房时应确保平层设计，货梯和机房应当位于同一水平层面之上。两者通路之间不应设有楼梯、门槛或坡道等建筑结构，以为设备运输提供便利与安全。有的数据中心的机柜高于机房地面，而且采用了阶梯的方式，这将给运输大型设备带来不小的麻烦和隐患。如果运输途中出现了坡道设计，一定要特别注意。坡道和水平面并不属于同一个建筑结构体。小推车最大承重大多为  $300\text{kg}$ ，因此要关注坡道的载荷承重。其倾斜角度不宜超过  $10^\circ$ ，以免在运输途中发生设备滑落，从而导致人员受伤和财物损失。





在机房内，要注意查看通道的宽度。最好是能够顺利地平行通过两辆小推车或移动操作台，并且留有一定的余量。平日工作时，我们经常会在某个通道内遇到其他工作人员，通道空间不足，在对向行进的时候就会撞车，势必要有一方退出去才可以。留有足够余量也是十分必要的，以免意外碰到外露的光纤或者突出的设备电源开关。

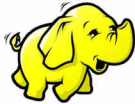
此外，机房内尤其是机柜上方是严禁有水的，凡是与机房无关的给排水、雨水管道等均不应在房间内穿越。

#### 4. 机柜的选择

在进行机柜的考察时，要检查以下问题。机柜的钢板厚度是否合规？柜门关闭后是否会留有较大的缝隙？机柜的进深是否足够（服务器设备应选用 800mm 以上的）？有些设备进深很大，如果选错了机柜，后期上架导致关不上门就尴尬了。测量的时候米尺和游标卡尺是必备的，如果你有现成的设备，直接上架看效果也是一个简单有效的好办法。

### 运维故事 2：奇怪的上架方式

Freedom 公司最近又成立了一家分支机构。为了尽快拓展业务，新机构正在紧张地筹建数据中心。由于总部之前刚刚和一家数据中心签订了合作协议，考虑到节省资金和资源合并，IT 部门的负责人希望和总部使用同一家数据中心。经过几次考察，他认为总部的需求方案和自己部门很类似，除了自己的空间利用率比总部高一些外，基本上没有什么太多的差别，于是双方很快签订了合同。这家数据中心的价格很低，所以他们只提供网络 and 机柜，其他服务是没有的，包括设备上架所需的材料。所以开始动工之前，部门还需要采购一大批用于设备上架的托条。在托条选购的当天，负责服务器的工程师小李被叫上一起去看现场考察。小李是第一次来这家数据中心，之前的考察并没有通知他一同前往，他只知道数据中心提供 43U 的机柜。由于过去团队一直秉承机柜高利用率的原则，按照以往的经验，这家数据中心的供电能力可以支撑 15 台 2U 服务器。但当他到了现场看到机柜以后却愣住了。原来这家数据中心的机柜和别处的不太一样，它的进深比较短，从机柜两边放下来的强弱电线缆，多余出来的部分无法放到机架后端，否则关不上门。这个机柜底部设计了一段横梁，直接占据了最下面的 2 个 U 位，这显然是为了盘多余线缆用的。说好的 43U 平白无故地减少了 2U 不能使用，再除去留给网络设备和理线架的位置，只剩下 35U 了。更要命的是这个机柜使用的是标准 U 位（44.45mm），而不是以前常用的非标 U 位（46.45mm）。因为托条本身有 2mm 厚，使用这种机柜显然是不可能实现全部设备上架了。这样一来，需要重新计算机柜的需求量，至少需要增加 50% 的开销。用导轨的话，上架速度慢不说，之前也没有这方面的预算，还需要财务重新审批，项目进度等不起啊。这样一个细节的忽略是非常致命的，但当时考察的时候没有人考虑带上一线的工程师去现场确认，现在再说这些都无济于事了。



有个工人提议，托条是可以稍微有一些形变的，三个一组是6U，5组共计30U，加上每组各留一个空位正合适。但是拿了三台设备进行实际测试的时候，发现如果是两台一组还可以，上下两个设备还可以略作调整，但三台根本行不通，中间的设备已经被死死地挤在里面了，根本动弹不得。又有一个人说，要不然干脆就上一根托条，然后把三台设备摆在一起得了。这可真是“好主意”，众人都没好气地白了他一眼。

最后还是施工方老雷出了个主意，确保不用增加预算，也不会强行挤压设备上架，方案还是使托条上架，每个设备两两一组，两台服务器之间多了一根托条的厚度（2mm），这2mm是不会对设备造成太大的受力影响的。而每组之间要有空位，但不是空一个U位，而是空一个孔。15台2U服务器共计30U，两两一组共分为8组，再加上8个孔所占的U位，最后还有富余。上架完成后，小李对老雷表示感谢：“老哥这活儿辛苦你们啦。”老王呵呵一笑：“咳，只要不影响施工进度，费点儿劲算啥？虽然不太标准，但不仔细看也没啥毛病。哈哈。”

谁说没毛病？空着的那些缝儿没法上盲板，会影响气流，制冷效果肯定不好。小李只是自个儿暗自琢磨，没好意思扫了老雷的兴。还好这家数据中心从来不考虑节能问题，冷风开很足。哎，就这么着吧。

## 5. 辅助区域的选择

数据中心提供的工位数量应当大于两个，并接入百兆以上的共享带宽。另外，如果机房工作比较繁重或者有7×24驻厂值守的场景，可考虑申请独立的小型办公区或者休息床位。

## 2.3 基础设施评估

如果把空间环境评估比作“看风水”，那么基础设施的评估工作可以算得上是“家装考核”了。基础设施系统主要包括电气系统、空调系统、消防系统、弱电及综合布线系统等。

### 2.3.1 电气系统

电力是数据中心正常运转的动力之源，电气系统的重要性自不必说。如图2-1所示，这是一个简明的供电系统示意图。数据中心的电力由外部引入，当外部供电出现故障的时候，则需要利用柴油发电机作为备用手段，对数据中心进行持续保障供电。

我们对用电有三个基本要求——可靠、安全和容量。

#### 1. 可靠性要求

用电的可靠性要从如下几个方面来考虑。

首先，输电侧的高可靠性。输电侧的可靠性是重中之重，每年数据中心因停电引发的大面积宕机事件并不是个例。引入一类市电可减少类似事故的发生。



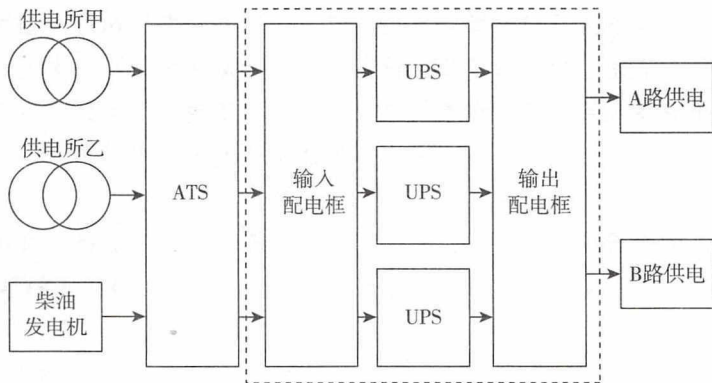
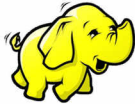


图 2-1 供电系统简易示意图

其次，全路径上的所有节点要保证充分冗余。说起冗余，我们作为运维人并不陌生，这也是老生常谈的话题了。所谓全路径，指的就是从输电侧到用户侧的整条输电线路。全路径上的每个环节都不能出现单点故障。这是一个逻辑与的关系，任何一处出现单点，整个系统的冗余就是零，跟没做一样。单条线路上的负载能力都应满足全部电力消耗总成，并留有一定的余量，一般应控制在 80%~85% 之间。在这里，经常会出现  $2N$  或  $N+X$  的概念。你可以简单地将  $N$  看成一个单点， $2N$  就是双倍冗余，等效为存储中的 RAID 1。而  $N+X$  则是指，在多组节点中灵活配置的若干冗余，即可以看作 RAID 中的 Spare 盘。

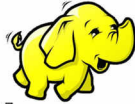
再次，就是备用方案的持续支援能力。一旦主线路失效，作为最后一道防线的备用方案，除了切换及时以外，还要有强大的持续支援能力，以应对主线路无法及时修复的情况。例如我们前面提到的天津港大爆炸，它所引发的断电故障在短时间内肯定是无法恢复的。此时的首要任务是进行切换。假设极端情况下切换失败，那就只能靠备用方案来扛了。

最后一点也很重要，基础设施的采购一定要选用知名品牌，这样才会更有保障。另外，选用产品的原则是求稳不求新。对于一些新兴的技术，即便它来自于知名厂商的产品，我们还是应当慎重选择。

## 2. 用电安全要求

除了可靠性以外，用电安全也是同等重要的。供电设施在长期使用后，会因线路老化发生漏电和人员触电的危险。因此，对于机房低压配电系统要求采用 TN-S 系统。什么是 TN-S 系统呢？在详细解释之前，我们需要先了解一些电的基本知识，看一看人体触电是怎样形成的。

电流和水的特性非常相像。常言道“人往高处走，水往低处流”，电流也是从高向低流动的。这里所说的高和低指的是电势。我们人为地将大地的电势定义为 0，平时大家所说的  $\times\times$  伏电压，指的就是输电侧相对于大地的电势差。除此之外，电流还有一个特性：如果遇到多条可以流经的路径，那么电流中的大部分电子总会选择电阻低的那条路径通过，即



常说的电阻和电流成反比。这和我们平时开车是一样的。电阻意味着交通状况，谁都愿意走行驶通畅的道路。如果有两条路径，一条路径的电阻是  $1000\Omega$ ，另外一条路径的电阻是  $9000\Omega$ ，那么这两条路径上面的电流比则是  $9:1$ 。

发生触电需要两个条件。首先要具有电势差，其次要形成有效的回路。我们经常看到小鸟能站在高架电线上。那是因为鸟的双脚落在同一根电线上，而鸟的个体很小，两脚之间距离所形成的电势差几乎可以忽略不计，所以它们才能够安然无恙。但是，如果一个人不小心摸到了电门，由于人站在地面上，而大地电势为  $0$ ，电流会因为电势差的关系，从输电侧经过人体流向大地并形成回路，这样就发生了触电。

为了防止发生触电危险，需要实施接地防护。以电源插座举例，插座面板上的三个孔分别为：左零，右火，中接地。火线（相线）是从输电侧引出来的线。零线（N线）由变压器二次侧中性点引出，用于和火线形成回路。电力在进入用户侧之前，由三根各成  $120^\circ$  的相线组成。理论上，三相之间应当是平衡的。此时 N 线不带电，但这种状态不可能一直保持下去。三相失去平衡时在 N 线上会产生电流，所以 N 线必须接地，否则当其断开时会发生触电危险。地线是直接接地的保护线，我们称之为 PE 线。因为 PE 线的电阻非常小，相较几千欧姆的人体电阻来说要小得多。一旦人体触电，大部分电流会被它引走，从而确保人员的安全。PE 线和 N 线的接地形式不同。PE 线是从变压器中性点接地后引出主干线，并每隔隔  $20\sim 30\text{m}$  重复接地。N 线则是在变压器中性点处与 PE 线重复接地，从而起到了双重保护的作用。

所以，接地是保证用电安全的主要手段。这里产生了一个有趣的问题：既然 PE 线和 N 线都要接地，那么如果我们把两股线合并在一起，岂不是节省了一大笔钱吗？这种系统就叫作 TN-C 系统，我们把这段合并的线称作 PEN 线。使用 TN-C 系统可以降低成本，但是它存在一些隐患。当 PEN 线路受损折断后，断点后端的设备金属外壳都会带电，此时如果有人不小心碰到就会触电。TN-S 系统的 PE 线和 N 线是完全分开的，安全性高了很多，当然成本也就随之提升了。因此在实际生活当中，人们大多采用折中的 TN-C-S 系统，即起始路径采用两线合并的方式，在入户之前将 PE 线和 N 线分开，而且不允许再次合并。

根据国际电工委员会 IEC 的定义，供电系统分为三类（五种）——IT、TT、TN（包括 TN-S、TN-C、TN-C-S），具体解释如下。

（1）第一字母表示电力系统的对地关系

□ T——中性点一点接地。

□ I——所有带电部分与地绝缘，或一点经阻抗接地。

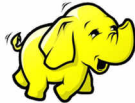
（2）第二字母表示装置外露可导电部分对地关系

□ T——外露可导电部分对地直接电气连接，与电力系统的任何接地点无关。

□ N——外露可导电部分与电力系统的接地点做直接电气连接（在交流系统中，接地点通常就是中性点），TN 系统后面还会有第三个字母。







### (3) 第三个字母表示中性线和保护线的组合关系

- S——中性线 and 保护线是分开的。
- C——中性线 and 保护线是合一的 (PEN 线)。

现在大家明白了吧。PE 线和 N 线分开与否，以及接地点的数量决定了安全等级和成本的高低。尽管选择 TT 系统更加安全，但它的成本过于高昂。建筑物内如采用 TN-C-S 系统，其前段 PEN 线上的中性线电流所产生的电压降会产生电势差，对设备造成信号干扰。而 TN-S 系统适用于内部设有变电所的建筑物。因此我们要求机房内的低压配电系统必须采用 TN-S 系统。

除了人员的用电安全之外，我们在选取机房的过程中，还要注意静电对设备的危害。以下这些内容是在实际机房选型中对接地和防静电提出的一些要求。

- 机房接地系统必须满足人身安全及运营设备正常运行的要求。
- 机房内所有设备的可导电金属外壳、各类金属管道、建筑物金属结构等均应做等电位联结，不应有对地绝缘的孤立导体。机柜应有两根不同长度的连接导体就近与等电位联结网格连接。
- 机房所在大楼的接地电阻应小于  $1\Omega$ 。
- 等电位联结网格应采用截面积不小于  $25\text{mm}^2$  的铜带或裸铜线，在防静电地板下构成边长为  $0.6\sim 3\text{m}$  的矩形网格。
- 机房地板或地面应有静电泻放措施和接地构造，防静电地板或地面的表面电阻应为  $2.5\times 10^4\sim 1.0\times 10^9\Omega$ ，且应具有防火、环保、耐污、耐磨性能。

---

## 运维故事 3：两路电

“唉，邮件又来了！”最近管理员小王有点烦。一周之内，这已经是他第三次收到数据中心张小姐“善意”的邮件提醒了。大意是说，小王公司的 300 多台服务器一直采用单路用电（我们专业的电源策略应当叫作 Redundant）的策略，导致数据中心机柜的 A 路高达 18A（最高供电 20A），而 B 路却是空载的。张小姐极力“建议”小王把现有服务器调整成两路均衡（Not Redundant）的模式。小王心里不太高兴，因为服务器默认出场配置都是冗余模式，又没有超电，为什么要我修改呢？刚刚休假回来的师傅老王听说这事儿以后，不一会儿就把所有服务器的电源策略调整为非冗余模式，并且给张小姐回复了一封邮件，告知其已经修改完成，同时对对方的提醒表示感谢。小王有点儿想不通，经过师傅的一番指点后终于明白了。“小王啊，人家机房建设多年，都像你这样高负载地总用一路，长期下来线路老化得快啊。张小姐一再催促你，她肯定是怕万一出了事儿，自己担不起责任。这就跟像穿鞋一样，两双鞋换着穿肯定比只可着一双穿要强。两路均衡的使用方式还是比较环保的，再说改个电源策略也很简单。你要学会相互体谅，理解万岁嘛。”

---



### 3. 电力的容量要求

电力容量是另外一个要关注的内容。容量选择和服务器选型有很大关系。以北京为例，一个机柜的年租用成本在几万元到十几万元不等，如果采购数量很大，这笔费用是相当可观的。因此提升机柜的利用率，是直接降低成本的有效手段。作为主数据中心，我不推荐采用 20A 以下的机柜。这样的话，空间利用率不会超过 70%，甚至会更低。企业规模越大越倾向于高电机柜的选择。如果当地电力设施条件允许，40A 以上可能是最好的选择。以笔者实际工作中的测试数据为例，单台 1U 服务器的电流平均值常年维持在 0.8~1A，峰值不超过 1.3A。这样算起来，40A 可以部署 30 台左右的 1U 服务器，再加上网络设备，无论是空间还是电力容量都实现了高利用率。PDU 插座的数量要和电力容量以及上架设备的数量相匹配，并且应当留有独立于机柜 PDU 线路的插座，以供显示器、能耗仪等外接设备使用。而外接设备不应当直接接入生产的 PDU 插座，以防止因外接设备自身问题导致的电力故障波及生产。另外，我要提醒大家注意的是，PDU 插座制式要符合国标。英制插座（俗称大头）在如今一些数据center里还是能够看到的。

#### 2.3.2 空调系统

空调的主要作用是降温和除湿，要想研究好空调系统，首先要搞懂气流。

说到气流，它可是一门挺复杂的学问。空调系统将空气进行制冷后，由风机送出冷空气，在与热源进行冷热置换后，达到降温的效果，经过置换后的热空气还要回流到空调系统再次进行制冷，如此周而复始反复循环。

那么要想达到好的制冷效果，控制好风速和风压是十分重要的。风速和风压是一个定值，它们是由设备机组决定的。我们这里把风压称为全压，全压总共由动压、静压和损耗三部分组成。气体在流动中会产生动能，这个动能所转化的压力就是所谓的动压。动压的大小和速度是成正比的。由于冷空气从风机里刚吹出来的时候速度快、动压大、气流不稳定，这些因素都会导致损耗增加。根据能量守恒定理可以得知，损耗越多，制冷效果肯定就越差。所以系统中都需要使用静压箱来减少动压、稳定气流，它可使送风更加均匀，从而提升制冷效果，同时还有降噪的功能。

电力和空调两大系统可谓是数据中心的任督二脉。甚至于有人这样讲：数据中心的核⼼就是管线和气流。然而它们却是一个看不见，一个摸不着。表面文章谁都会做，但真正的秘密其实都隐藏在那些看不见的地方。

### 运维故事 4：地板下面的秘密

老张最近忙着数据中心扩建的事。招标书刚发出去没两天，各大数据中心的销售纷至沓来，可真是快把他办公室的门都挤破了。这不，今天老张又被一家数据中心请去参观。接待老张的是销售经理 Cindy 和售前技术经理 Eric，一番寒暄后，先是由 Cindy 介绍了一

下公司的情况，接着在 Eric 的引领下，大家一同去实地进行了参观。整个机房内窗明几净，整齐划一，布局合理，标准规范。老张看了看，不住地点头：“贵公司的数据中心建设得非常规范，简直可以做样本典型了啊。”

“哎呀，你过奖了，如果你觉得满意，我们在价格上还可以给你一个更加优惠的方案。” Cindy 听到老张的赞许，迫不及待地插话进来。

“不过，能不能掀开几块地板让我瞻仰一下？”

Eric 微微一笑，“请你稍等”，说罢便转身走出了房间。不一会儿 Eric 一脸为难地进来了，“抱歉啊张总，实在是凑巧，掀地板的地吸被工人拿去施工了，你看……”

“不妨事，要不我们直接去工地瞧瞧？”老张不动声色地说。

“哎呀，你是大领导，工地乌烟瘴气的，怎么好亲下现场呢？我们去会议室休息一下吧。”Cindy 也在一旁劝说。

“哈哈，算啦，今天也不早了，我也该回去了。整体来说确实不错，我回去向老板做个汇报，具体的我们会认真考虑的。”

一行三人来到大门口，临别前 Eric 不断地恭维着：“张总你可是老前辈，这次和你交流，我也是长了不少见识呢，回去之后还有劳你多多美言啊。”

“哪里，哪里，Eric 老师你也是个中高手啊，哈哈。”

这番对话在 Cindy 看来，不过是两个人在相互恭维的商务礼节而已。但她却没有注意到，Eric 说这话的时候，他脸颊上的微红与额头沁出的汗水。

### 1. 送风模式

送风模式根据送风出口的位置主要分为两种。

第一种是上出风模式，如图 2-2 所示。它类似于日常生活中的空调系统。由于其送风方向和冷空气下沉的特性相一致，因此它的局部范围制冷快、效果好。此外，上出风模式还有不易产生积灰、设备问题易排查、扩建方便等诸多优势。然而它的缺点也非常明显，因为热空气往上走，而冷风往下送，送风方向和空气流动特性相悖。也就是说，这两者相互“顶起了牛”，使得机房的最下部区域的温度相对偏高。另外，由于屋顶的空间有限，对于进深比较大的机房需要增加送风管来保证送风的均匀，噪声也相对比较大。在承受辐射和寒冷的基础上，如果再加上噪声，这种环境对于在机房中长期工作是不利的。

第二种是下出风、侧回风模式，如图 2-3 所示。这是现今大多数新型机房采用的送风模式。因为下出风顺应了空气流动的特点，而且地板下方空间所

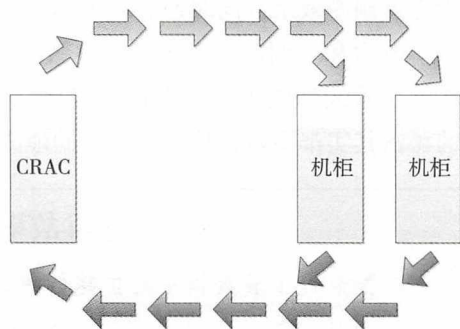


图 2-2 上出风模式

构成的静压箱要比风管的横截面大得多。因此同等条件下,下出风的整体送风更均匀,温差更小,制冷效果更好。而且施工简单,无须风管和送风口。但下出风也十分考验地板材质、施工质量和布局的合理性。如果产品或施工质量不合格,会因为漏风出现送风短路。再有,地板下面容易积灰。防尘做不好的话,风机一启动就是漫天飞尘。建筑的层高不足,也不应该强行使用下出风的模式。因为活动地板空间有限(例如高度 $<400\text{mm}$ ),后期扩建时的管线增加,肯定会影响到送风效果。还有,由于管线被埋在地板下面,不利于故障排查。这些都是前面故事里提到的地板下面的“秘密”了。

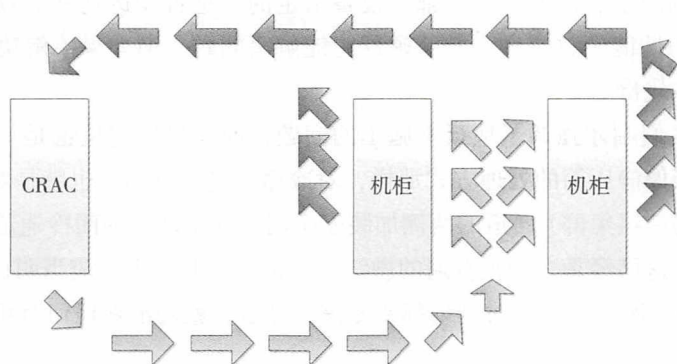


图 2-3 下出风、侧回风模式

我们观察地板下的施工情况时,可采取一些简单方法。看走线是否规范,管线空间是否足够,布局是否合理,防尘保温做的是否到位,等等。具体指标可参考国家的标准规范,里面都有明确的指导。我们这些行外人不可能做到一一深谙其道,但至少你可以找到几条便于理解和判断的标准,暗暗牢记在心,到现场有目的地去考察。另外,可以多咨询一些专家,还要多看多问。其实很多知识往往就是在技术交流的过程中慢慢学习并理解的。

服务器风扇如果吸入的制冷量不能满足要求,会产生局部过热的现象。大数据的应用和强调高性能计算的今天,很多企业在选购服务器的过程中,也在大量选用高性能 CPU 和大容量磁盘的硬件配置。大功率和高密集型服务器的出现,已经向传统制冷提出了新的挑战。

为了保证充分有效地冷热交换,使用更高功率的风机显然是不节能的。而水平送风技术为我们提供了一种新的解决方案。水平送风的架构是制冷机组紧贴机柜安装,由多组风扇将冷空气水平送出,之后被吸入服务器,完成冷热交换后,再由服务器风扇排出,彼此之间就形成了一个有效的气流循环。尽管水平出风在我国并未真正普及,但这个思路还是值得我们学习和思考的。如果这种方式和下出风模式相结合,也许在不久的将来就会广泛地应用起来。



## 2. 密闭问题

在机房里，我们经常听见别人谈论冷通道和热通道，顾名思义就是冷空气和热空气流动路径所构成的空间。以主流的下出风模式举例，我们看到现如今的机柜码放方式都是采取“面对面、背靠背”的策略。两个机柜正面之间的夹道就是冷通道，冷空气从通道下方送上来，从前方被吸入，再经置换后被风扇从机柜后方排出。而机柜背面之间的夹道就是热通道，热空气在此聚集后向上蒸腾回流到机组当中。

一般人们认为只要有足够的制冷量，使得温湿度达到要求就万事大吉了。但却忽略了气流的重要性。制冷量、风速和风压都是设备给定的，是否能达到好的效果是由气流决定的。这就和系统的性能调整一样，升级硬件也能解决问题，如果不解决关键因素，势必付出高额的成本代价。

关于密闭，我们刚才强调了地板下施工的问题，而在机柜这里也是一样。为了确保制冷效果，我们应当像静压箱的处理方式那样，对冷通道进行密闭。也就是对模块（两排面对面的机柜所形成的一组集群）通道的两侧加装安全门。除了起到封闭冷通道的作用，也增加了门禁的安全性，这已经是一种很普遍的做法了。讲到这儿，大家应当明白：送风短路（漏风）是制冷的大忌。所以我们时时刻刻都要考虑一件事，就是消除任何有可能发生破坏气流的因素。

**第一，空闲机柜的影响。**假设一开始我们采购一个模块，但并不是所有的机柜都加电使用（因为加电意味着正式使用是要付费的），而是随着设备采购，逐步地开放机柜使用。空闲机柜如果没有加装盲板，由于空机柜没有设备，灌入的冷空气既没有参与热交换，也无法有效地形成回流，破坏气流平衡，造成能源损失。

**第二，托盘上架方式。**我刚入行那段时间，接触过一些企业的内部机房。当时设备上架，像小型机这种贵重设备都是导轨上架，而 x86 服务器为了省事儿，都是直接将设备安置在托盘上的。托盘的种类分为两种，有上螺丝的，也有插销式的。因为托盘拆装方便，又比导轨的通用性好，所以这种上架在早期还是很流行的，当时很多整机质量较轻的设备都用托盘。但是为了防止形变，托盘面板都比较厚，因此是不能实现设备紧密码放的。很多人都习惯在上架的时候，各个设备之间留有 1U 或者 2U 的空间。现在大多设备都是前进风后出风，风扇足以带走大部分的热量。所以没有必要再留有空余。这样做既无益于散热，又浪费了机架的利用率。因为托盘的厚度会挤占下面 U 位的一部分，托盘和设备之间会留下不足 1U 的空位，而这段空位无法上盲板进行密闭，非常影响气流循环。现代数据中心采用托条来完成上架工作。它既继承了托盘的通用性优点，又实现了设备紧密摆放的需求。而那些没有设备的空 U 位，则需要加装盲板实现机柜密闭。

**第三，机柜设计问题。**比如机柜边缘或对接处有缝隙。像这种有先天问题的机柜最好敬而远之。如果已经不可避免，也可以采用密封条等手段来补救。



### 3. 精密空调的选择

在充分满足机房安全的条件下,应尽可能地选择高效节能的空调系统。空调系统常见的有风冷和水冷两种方式。说到制冷,最难熬的莫过于炎炎夏季了。这个时期是用电的高峰,最怕的就是停电。风冷系统需要定期换气,从外部引入新风与回风进行混合。所以,水冷效果显然在夏季要比风冷更好,而风冷的效果在冬天会得到加强。当然,选用水冷系统要看实际的地理位置和使用年限。水冷的效果受到水质的影响,而且后期维护成本高,时间一长制冷效果会越来越差。如果你是新机房,水冷肯定是优先选择的。如果当地水质不好,或者制冷机组已经用了很多年,可能就要实地考察一下制冷效果了。水冷一般都会选择下出风,万一你真遇到了上出风的水冷,一定要考虑漏水是否会影响到设备的安全。

精密空调的安装位置也有讲究,应尽量靠近热通道和热负荷区域。在设备选型上,应尽量选择单台制冷量更大的设备,并且以显冷量作为主要衡量目标,而非全冷。这里涉及两个概念——显冷和潜冷。这两个概念是由热力学中的“显热”和“潜热”派生而来的。具体的含义就不做过多解释了。简而言之,我们知道空调有两个功能——温度调节和湿度调节。所谓显冷量是指降温时所需的冷量,而潜冷量则是指产生冷凝水所需的冷量。全冷=显冷+潜冷。而数据中心的首要任务是降温,因此在选型过程中应以显冷量作为主要衡量目标,以免低估了制冷量的需求。

精密空调的冗余方式为 $N+X$ 。这里 $X$ 的值我们推荐最好是2或3,太少可靠性差,太多浪费成本。关于 $N$ 的显冷总量的设计,我们建议机房最大的总负荷值不应该超过显冷总量的85%。

### 4. 气流组织

数据中心对气流组织的要求主要包括以下内容。

- ❑ 冷通道内温度任意点均不高于 $24^{\circ}\text{C}$ ,热通道温升对比冷通道不超过 $12^{\circ}\text{C}$ 温差,避免通风量不足造成局部过热;针对高功率机柜散热,应通过增加通风地板面积、通风孔率,局部风扇地板,局部精密空调等方式解决。
- ❑ 在空调因断电或其他因素停机或切换时,机房冷通道内温度不超过 $30^{\circ}\text{C}$ ;一般情况下,5kW机柜,末端空调恢复通风供冷时间不超过10min,8kW机柜,末端空调恢复通风供冷时间不超过6min,否则机房会出现规模过热的风险。
- ❑ 板下送风气流组织,下送风应均匀,风速不超过 $3\text{m/s}$ ;采用风道送风时,送风风速应满足相关通风风道设计规范,主干管风速不超过 $8\text{m/s}$ ,支管风速为 $3\sim 5\text{m/s}$ 。
- ❑ 送风距离:单侧送风距离应小于15m,大于15m应双侧送风。

## 2.3.3 消防系统

数据中心是耗电大户,因此必须加强防火意识。机房耐火等级不应低于二级。机房应



当配备至少两个以上的安全出口，门的开启方向要和疏散方向保持一致，且在任何情况下都应当能够从房内开启，并完成自动锁闭。灭火装置应选用 IG541 或者七氟丙烷气体。表 2-1 是这两种气体的特性对比。

表 2-1 IG541 和七氟丙烷的特性对比

| 特 性           | IG541                | 七 氟 丙 烷              |
|---------------|----------------------|----------------------|
| 理化特性          | 无色无味、不导电、无腐蚀性、无污染、无毒 | 无色无味、不导电、无腐蚀性、无污染、低毒 |
| 存储压力（20℃）/MPa | 15、20                | 2.5、4.2、5.6          |
| 传输距离 /m       | 150                  | 30、45、60             |
| 存储形态          | 气态                   | 液态                   |
| 部署条件          | 有管网                  | 有无管网均可               |
| 灭火原理          | 物理降低含氧浓度             | 化学反应降低可燃基团物理降温       |
| 优势特点          | 更环保，输送距离远            | 单瓶储气多，省空间，效果更好       |

2.3.4 弱电与综合布线系统

弱电系统主要包括传感器监控、影像监控、门禁系统及入侵检测等。总体原则上只要做到无死角即可。除了空调机组自带的回风温度探头，在机房的每个冷通道内，至少头部、尾部和中间应各部署一个温度探头，在热通道内部署一个温度探头。每个通道两侧应各部署一个摄像头。

机房内应当有不少于两部可用于外拨的固定电话。通常，机房内是没有手机信号的，因此要部署固定电话，以便在遇到特殊情况时，能够直接和外界及时联络。

如果你比较关注机房的总体情况，例如能耗、温湿度等信息，希望能够将相关数据直接接入自己的监控系统，那么机房的监控系统需要支持类似 HTTP、SNMP、TCP/IP 等主流的通信协议。

综合布线的大致要求有三点：第一，强、弱电的线缆要分开，并且要注意它们之间的间隔；第二，线缆走线要平行无交叉，途经路径要注意规避潮湿、易漏水、易喷水等地方，该套管的地方要套管；第三，管线空间规格要符合标准，注意有可能折损、破坏线缆的因素。

以下这些内容是我们在实际机房选型中对综合布线提出的一些要求。

1) 强弱电分离：机房桥架应满足强电线缆、弱电线缆、光纤等分开布放需求；一般情况下，机房应采用上走线、下送风方式，走线架宽不小于 400mm，下层走弱电，上层走强电，弱电桥架内安装光纤槽道，强电与数据铜缆、光纤之间的间隔不小于 200mm；当采用开放式走线架时，如果弱电走线架与强电走线架平行布置，则二者的间隔距离不应小于 300mm。



2) 光缆槽道: 普通机架光纤槽深度 100mm, 宽度 120mm, 核心网络区域机架光纤槽深度 100mm, 宽度 320mm, 出线口要求不小于 300mm。机房内所有机架要求普通光纤槽覆盖。

3) 机房应保证提供双路由光缆接入条件, 允许其他运营商光缆直接入局或与其光纤互通, 并协助解决相关光缆接入事宜; 具体光纤需求数量配置能满足按需扩容; 当有到达同一站点不同路由的需求时, 可通过迂回绕接来保证全程无路由重叠, 光缆落地施工时务必尽可能走不同的管接进进机房。

4) 光缆技术指标: 线路光纤类型 G.652, 平均单位光功率衰减常数  $\leq 0.5\text{db/km}$ ; 光纤色度色散系数应  $\leq 18\text{PS/km} \cdot \text{nm}$ , 偏振模色散  $\leq 0.2\text{PS/km}^{1/2}$ 。

5) 连接可用性: 丢包率不高于 1% 情况下, 保证连接可用性达到 99.99%。

## 2.4 网络建设评估

这里所说的网络建设, 不是指具体的网络性能指标, 而是指数据中心能够提供的网络资源与建设资质。数据中心的网络资源有专线和裸光纤两种。专线传输距离远, 但租用价格高, 带宽上有限制。裸光纤理论上带宽不再受限制, 适合短距离、大数据量传输。同城多机房的各节点之间互通建议使用裸光纤。但拉裸光纤这种事吃力不讨好, 造价高利润少, 作为运营商, 肯定是不大会随便开这个口子的。如果这些节点不是同属一家数据中心, 那么它们之间要实现互通, 可能就更加困难了。这个时候你就知道, 一个有着丰富人脉资源的数据中心经理是多么重要了。

前面我们反复提到过冗余问题。在全路径的各个节点上, 都需要做好冗余结构才行。假设在北京, 从东四环到南五环外路穿线。正常情况下, 一条应该走四环, 另外一条走五环才对。不能所有的线缆都走同一条路径。再如, 铺设线缆应该进入正规的管井之中。如果承建商根本没有资质和资源, 被管理部门拒之门外, 最后只能把部分线缆甩在外面, 也就是我们常说的“野纤”。野纤再加上单路径, 其隐患可想而知。之所以会有这些问题, 归根到底就是一个“钱”字。资质要花钱, 多路径也要花钱。而这些工程隐患的出现, 正是一味追求低价所引发的后果。

## 2.5 服务保障评估

可用性要考评数据中心的对外服务能力。据说, 200 个机柜对一个 IDC 运维来讲, 他的工作将达到饱和状态。数据中心是不太可能提供过多的人员充分保证每一家用户的使用需求的。如果有 7×24 小时值守服务的特殊需求, 应确保有专职人员服务, 在合同需求中也要有所体现。如果数据中心保证不了, 则需要另行招聘驻场人员。

## 运维故事 5：一山难容二虎

销售经理 Tina 听说 Forever 公司要扩建新的数据中心，急忙和售前经理赶去拜访。她精心准备了几十页的 PPT，不遗余力地为自己的企业进行宣传。当讲到服务客户这一页时，Tina 显得很有自信：“我相信贵公司如果选择我们的服务，那将会给你带来更加有效的保障。因为我们正在为很多知名企业提供数据托管服务，例如 Something、Balabala，还有……”

深谙市场行情的老 D 忍不住插了一句：“抱歉打断一下，我听说 Never 好像也在你们那里哦。”

“啊？呃，是的。”这个突如其来的问题让 Tina 显得有点儿不知所措，她刚刚才发觉到 Forever 和 Never 的竞争关系。

老 D 又补了一句，“我们的需求数量和 Never 在贵公司现已保有的数量相近，大家都是大客户，那么请问，如果出现资源争用的情况，你们会优先保证谁的服务呢？”呃，这句话没法接啊，一时间 Tina 自己都不知道该说些什么好。

## 2.6 本章小结

上述我们所讲的不过是数据中心的一些皮毛而已。具体的细节，大家可以参考以下这些规范（因书籍出版的时效性，请读者自行更新最新版本）：

- 《计算机场地通用规范》(GB/T 2887—2011)；
- 《计算机场地安全要求》(GB/T 9361—2011)；
- 《信息安全技术 信息系统安全等级保护基本要求》(GB/T 22239—2015)；
- 《数据中心设计规范》(GB 50174—2017)；
- 《数据中心基础设施施工及验收规范》(GB 50462—2015)；
- 《建筑物电子信息系统防雷技术规范》(GB 50343—2012)；
- 《气体灭火系统设计规范》(GB 50370—2005)；
- 《气体灭火系统施工及验收规范》(GB 50263—2007)；
- 《综合布线系统工程设计规范》(GB 50311—2016)；
- 《综合布线系统工程验收规范》(GB/T 50312—2016)；
- 《视频安防监控系统工程设计规范》(GB 50395—2007)。

最后我们总结一下考察机房的几点注意事项。

第一，便宜没好货。在招投标里，商务价格往往占比很重。少数厂商为了中标，不惜赔本儿赚吆喝。然而，有些成本是省不了的。如果一味看重低价，甚至是不合理的低价。其结果势必会有两傻：要么乙方是傻瓜，赔钱做善事；要么甲方傻眼，到时候不是质量有

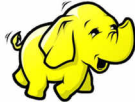
问题，就是后续要产生费用增项。

**第二，多看实际生产环境。**有条件的话，应当尽可能去生产机房参观。建议多在各个楼层进行走访，考察不应只停留在一个地方。

**第三，不要只提问不回答。**项目实施最忌讳的就是沟通不充分。甲方往往容易犯一个错误——那就是提问多，回答少。甲方总是关心乙方能不能满足需求？你想有谁会说自己满足不了呢？需求讲不明白，细节方面落实不到位，对方又没能充分理解，就容易产生偏差。能不能做不是重点，需求有没有充分地表达清楚才是关键。

当然，最后还是要多多学习，扩展自己的知识面。就像我开头讲的，可能公司没有那么多专职岗位，一些工作需要你兼职来完成。建议大家平时要多积累一些周边知识，做一名T型人才。





## 第3章

# 数据中心的规划设计工作

万丈高楼平地起。一个优秀的规划设计方案是一切良好的开端。数据中心的规划设计工作是分阶段的，并非一蹴而就。它的生命周期贯穿于整个项目建设的进程之中，我们可以依照时间点将它分成三个阶段。

**第一阶段——准备工作。**在开始规划设计之前，首要的任务是完成设备选型和能耗测试。设备选型应依据业务需求而定，方案确定以后，才能据此展开能耗测试的工作。能耗测试的结果非常重要，它是计算电力容量和空间利用率匹配最优解的核心参数。由于各家数据中心的电力容量和机柜价格并不相同，根据最优解，你才能够进一步得出总体成本的最小值。因此，设备选型和能耗测试可以帮助我们找出那些符合预期要求的数据中心。

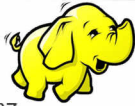
**第二阶段——实地调研。**在入围采购的筛选过程之中，实地调研是必不可少的一个环节。在这个过程中，我们要关注实际情况与最初设计方案之间是否存在着一些偏差。如果这些偏差不是硬伤，那么这家数据中心就具有入围资格，未来也有中标的可能性。设计方案就要为此做出一些调整。比方说，为了确保网络链路的冗余，我们希望单排机柜的数量以偶数为佳，便于将来布局时采取两两一组的形式来分配机柜。如果实际情况恰恰相反，那么单个机柜怎么用就是你要考虑的事情了。因此，在调研时最好能够参阅数据中心的平面设计图。从全局的角度观察，能减少问题死角带来的麻烦。从这一点上我们也能看出，设计方案存在着动态调整的可能性。

**第三阶段——平台建设。**在完成采购任务后，接下来就是进一步的细化工作了。这里的工作重点在于——设计方案要如何保证各个子系统之间的平衡。在规划具体的细节时，建议邀请网络和系统方面的一线工作者一同参与探讨，充分考虑他们在实际工作中的需求。此外，还要注意业务需求的多变性，它会带来很多不确定性的因素。

---

### 运维故事 6：计划赶不上变化

眼看年关将至，信息中心新一年的机房设备采购计划再一次启动了。因为业务增长得太快，现有的机房已经不够用了。为了确保日常维护工作的正常运行，经过上级领导批准，已经同意信息中心走内部的续采流程。信息中心将和现在这家提供托管服务的数据中心再



签订一份增补合同，把现在正在使用的306号机房旁边的308号机房也租用下来。新机房的规模和布局与老机房完全一致，就等各部门上报采购计划了。只要需求数量一落实，就可以确定二期扩容所需机柜的数量。

时间紧任务重，领导让新来的雷小虎负责本次统计规划的工作，要求各组负责人尽快上报自己团队的采购计划，把数量交给雷小虎汇总。过了一个礼拜，各组的数据陆续递交了上来。雷小虎看了一下数据清单，发现业务二组的数据还没有上报。于是，他赶忙去找二组组长老尤求助。

“尤组，上周领导吩咐我统计采购计划的需求数量，你看能不能安排个人配合一下？”老尤嘿嘿一笑，“这事好说，我让小米来支持你。”

“小米，你来一下。”

“领导，需要我做什么？”听到老尤的招呼，小米不敢怠慢，急急忙忙地跑了过来。

“中心需要各组配合申报明年的采购计划，你就负责配合小虎完成咱们组的任务吧。好了，具体的事情你们俩对一下吧。”

“好的，老大，我一定全力配合。”小米笑咪咪地连连点头。

“那就这样，一会儿还有个会，我先走啦。”说罢，老尤转身离开了。

小米看着老尤走远后，转过身来一副笑脸地对着雷小虎言道：“虎哥，关于这个具体问题呢，我也不太清楚。你看，我们日常工作也只是负责维护和项目对接而已，这个设备具体的使用方是国际业务综合办。说白了我们其实就是一个二房东，代人家提采购需求的。我看咱俩还是直接去找综合办的小胡问问吧。”

于是，两人找到了综合办的小胡。小胡介绍说，来年准备上一个新项目。因为综合办在四季度的走访调研过程中得知，一些用户反映社区业务的沟通渠道少，办事效率低。为了支撑社区工作开展，综合办决定设立一个论坛空间。但因为要给外网访问，不能和现有业务放在一起，需要单独隔离出来。大概需要五台服务器。

“只有五台啊，这可有点儿浪费呀！”雷小虎暗自嘟囔着。因为机柜都是两两一组，即便只用几台做隔离，也势必要挪出两个空机柜来。

“那你们以后会不会扩容呢？”雷小虎突然想到这个问题，有点不放心地又追问了一句。

“哦，这个应该不会吧。我看就算是扩容，也增加不了几台机器啊。”

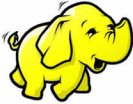
“好，就这么定了，那我就写报告去了。”看着小胡信誓旦旦的表情，雷小虎天真地相信了。

又过了一个月，小胡领着小米兴冲冲地跑来找雷小虎。

“虎哥，上次咱们讨论的那个社区论坛的项目需要再增加50台服务器，你看能不能尽快给我们安排一下。”

“什么？再加50台？你们不是说不再扩容了吗？怎么会又多出来这么多需求？”

“嗨，原来我们以为增设论坛只是为了满足一部分人的需求，所以就没太在意。没想



到这个论坛一上线，立刻就引起了很大反响。很多用户打来电话，说除了论坛希望能再增加一些什么短信提醒、邮件反馈之类的功能。用户热情高涨，领导说这个项目要加大投入，所以我们的需求就变多了。”

“可是你们这个项目是需要做隔离的，当初就作为一个特殊需求，没有预留那么多的地方啊？”

“那咋办？我们现在很急啊！领导现在是亲自督办此事，不能耽误啊。”小胡一脸不高兴的样子，转过身来问小米，“你就说我们现在去找谁吧。谁能解决这个问题？”

“你……”雷小虎不知道该说什么好。综合办是业务部门，在单位里那可是一线明星部门，很有话语权。而小胡一看事情不好办，有可能要耽误她的工作，脸立马就拉下来了。这种瞬间的态度变化让耿直的雷小虎有点儿接受不了，尤其是她那动不动就搬出领导压人的姿态更是让人觉得不爽。

小米在一旁开了腔，“据我所知，咱们不是还有空闲的机柜吗？”

“空机柜倒是有，可是那些都是我已经规划好了的呀。”

“但是你现在也没有别的办法呀。我们先临时借用几个，到时候买了新机框再还你几个不就行了嘛。唉，项目进度重要啊。”

“就是，就是。到时候我们加倍偿还。嘻嘻。”小胡也在一旁打趣到。

“你们知道什么，这又不是借橡皮，使完了还回来就行了。”这句话雷小虎嘴上没说出来，但心里很不痛快。借出去的机柜哪有原封不动地归还的，不过是拆了东墙补西墙罢了。虽然数量上是没有亏损。但是原来整齐的规划排序就全乱了。本来他已经将不同的区域都用颜色标识给规划好了。这样来回一折腾，计划好区域就打散了。好好的彩色规划图，弄得就跟马赛克壁纸一样。以后管理起来还是个大难题呢。

两周之后……

综合办的老白也来找雷小虎。

“小虎，我们最近又有一个新项目，需要 50 台服务器，而且要和现有项目隔离。你给看看怎么弄？”得，麻烦又来了。

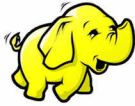
“白哥，你确定就是 50 台，你们还有没有扩容了？”

“这个我可说不好。你就先来 50 台吧，到时候不够再说。”

“白哥，都这么弄我可有点儿掌控不住啦。你不提前规划好，我不能总是东拆西借的啊。”

“扩容这事儿我哪能知道啊？我又不是诸葛亮能掐会算的。我现在就要 50 台，其他的以后再说。”话没说两句，老白反倒先露出了不耐烦的神情。“那好吧，白哥，我先去趟洗手间，回来咱们再商量。”说罢，雷小虎阴着脸走出了办公室。他哪里是上什么洗手间，现在的他气血上涌，只想找个地方一个人静静。





### 3.1 需求的不确定性

规划最怕的是变化。变化是我们在规划设计中遇到的最大的阻力与麻烦。问题的根源就在于需求具有不确定性，它就像是测量学中的误差一样不可避免。很多项目变更的出现，正是因为前期需求没有讲清楚造成的。“讲不清”的主要原因有两个：一个是“说不好”，另一个是“没想到”。这种问题在那些新项目中尤为常见，特别是在创业公司或者业务转型的时期及其普遍。

开展一个全新的业务，存在着很多不可预知的因素。我们对于新业务的整体运作是抱有试水心态的。大家都知道“小马过河”的故事。虽然在过河之前，小马通过调研的方式对可行性进行了充分的论证，但它毕竟没有实践经验，所以在迈出第一步时，是没有十足把握的。同样，新项目上线后，你并不能确保它就一定朝着理想的状态发展。这是一个尝试的过程。如果顺利，投资力度自然就会加大。反之，业务就会缩减甚至取消。

从这个逻辑角度上来讲，业务部门在很多情况下是没办法预测需求的。我们发现了一个有趣的现象，预估的需求量总是小于实际的需求量。这是因为申请人采取了保守策略的缘故。人们常说“不要把弓拉满”，采取摸着石头过河的方法，其实就是为了给自己留有一些余地。但不管怎么说，项目成功的可能性更大一些，所以增项变更是大概率事件，我们必须为此要做好一些额外的准备。如果你处理不好这种问题，将会给后续工作带来非常大的麻烦。

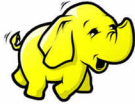
---

#### 运维故事 7：续采之怒

“他严涛到底是什么意思？这分明就是在要我！”采销科主任刘致远把桌子拍得咣咣直响。一看老大发了脾气，科室里面的人一个个全都低着头不敢吱声，生怕这倒霉的无名之火烧到自己的头上。

“这个采购单我刚刚结束还没有一个月，他又来提交续采申请。难道他不知道这种事情在审计那里是通不过的吗？”

单位有明文的规定，所有采购项目必须招标，是不允许直接采购的。当然，有的时候为了保证项目运行的连续性和可靠性，在说明理由的情况下也可以申请续采。但是，这次项目采购刚刚结束，供应商尚未供货。在这个时间就去提续采申请，审计显然是不会给予批准的。如果非要续采，意味着刘致远必须重新开启一轮新的招投标，相当于一件工作做了两遍。即便这样，审计处的人照样也会找他的麻烦，一定会问他什么要提两次采购需求。当然，他严大主任只是替国际业务综合办的老白提采购需求而已，他要的就是进度，其他事情他是不会关心的。所以刘致远觉得自己平白无故地背上这口锅实在是很憋屈。



就为这事儿，俩人在例会上针锋相对地大吵了一架。本来这事儿确实是不太正常，但各部门都有自己的理由，而且综合办的政治地位在单位里那是数一数二的。最后业务、审计、采销部门的几个高层领导一商议，批示如下：请采销科着手速办，请审计处绿灯放行。一切以业务为重，这种情况下不为例。

下不为例？鬼才相信呢。在刘致远看来，姓严的根本就是不负责任。像这种情况不是一次两次了。前期需求没搞清楚，出现了变更之后，也不和自己提前沟通一下，先斩后奏地搞突袭，最后再让综合办的人出面给他施压。再这么下去，以后的工作真是没法干了。

## 3.2 如何避免变化打乱规划

既然需求变更是不可避免的，那么提前预判并采取积极有效的应对措施就显得十分重要了。假设我们将需求比作一只吹满了气但没有扎口的气球，一旦松手，这只气球会到处乱飞，毫无规律可言。而规划设计就像是四周的墙壁，它会限定气球的飞行。如果墙壁包围的空间过小，气球在飞行的过程中就有撞墙的可能，这就代表需求和规划之间产生了冲突。如果我们能事先评估，尽可能地将所有的飞行路线都考虑进去，并为此预留出一些空间，那么冲突的概率就会降低很多。

### 3.2.1 采购资源预留

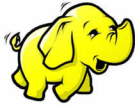
比如说上面这个续采的例子，业务部门是设备的使用者，但是设备资产和管理是算在 IT 部门头上的，所以 IT 部门要负责提交采购申请。这里就存在很多矛盾。东西不是我用，但我要负责提采购需求，而采购需求又说不清，怎么办？大额采购的周期是比较长的，不可能让你频繁下单。但预留太多资源，财务又会削减你的预算支出。假设业务部门计划采购 50 台设备，你在此基础上又预留了 50 台。财务部门在审核的过程中，可能会对业务部门进行调查。业务部门原本就认为自己只需要 50 台设备，在面对挑战时，没准儿还会缩减最初的计划。此时，这 100 台的设备采购预算肯定要遭到削减。

从这个例子上可以看出，资源预留不能闭门造车，必须和业务方进行充分沟通，就各种风险及解决方案达成共识。

首先，技术部门要提醒业务方，后续有可能出现增项变化。然后针对这一风险，给对方一个合理化的预留建议。比如把设备数量从 50 台改为 70 台。

其次，要注意需求有失控、超出预期的风险。如果预留资源在未来无法满足增项需求，我们将面临着两种抉择：要么砸掉原有的墙壁继续扩容，要么让气球停下来。无限扩容显然是不现实的，好的做法是拆解超出范围的那部分需求。假设设备增项是 50 台，我





们看一看在这 50 台里面，能不能把最紧急、最重要的先安排进去，其余的挪到项目二期去完成。

最后，还有一点要特别注意：在后期的扩容工作中，你要考虑是否存在迁移、停机等一系列问题。把这些内容明确下来，让业务部门有充分的心理准备，双方要就此达成共识，这一点非常重要。

### 3.2.2 数据中心机柜区域的规划与布局

设备的上架与迁移，在数据中心的日常工作中乃是家常便饭。如果前期没有一个合理有效的规划，加之人员变动频繁、岗位交接不清等因素的影响，日久天长容易出现资产混乱的问题。

古人云：一室之不治，何以天下家国为？机柜区域的规划与布局可是很有讲究的。你可别小看它，一个合理、高效的规划布局，不但能最大程度地提升空间利用率，节约项目开支，同时还具有很强的视觉效应。如果方案设计得合理，你完全可以将整张平面规划图清晰地印在脑海之中，便于理解和记忆。那么，日常的管理工作就会变得更加轻松。反之，如果你走到一处，都不清楚这里是干什么的，工作效率想必也高不到哪儿去。

按照应用的角度划分，我们可以把整个空间分为三个部分——生产区、非生产区和基础设施。生产区的设备均属于线上系统，只能由运维团队来管理。非生产区则主要用于开发和测试的工作。它对可用性的要求不高，但对自主操作的需求非常强烈。所以，非生产区的权限可以更加开放一些。由于开发和测试本身就有不确定性，在使用过程中会带来一些破坏，因此非生产区与生产区之间必须实施隔离管理。这里有物理隔离和逻辑隔离两种形式。前者需要在网络的拓扑结构上进行隔离，后者则可以通过防火墙来达到目的。数据中心的基础设施不作生产与非生产的区分。生产区和非生产区都有各自对应的基础设施，你可以将它们分别放置在不同的机柜里以示区别，但它们应当依旧同属一个空间。

接下来，我们还可以将这三个空间进一步细分成九种不同类型的区域。详细情形如图 3-1 所示。

下面，我们分别介绍这九个区域。

#### (1) 网络区

顾名思义，网络区主要用来安置核心层的网络设备，以及与银行对接的一些专线设备。该区域的 Owner 为 NE（网络工程师）。

#### (2) 管理区

管理区用于提供维护管理功能，它涵盖了所有的基础服务。例如，部署系统、资产系统、DNS、文件共享、配置管理、监控系统、安全检测系统等。该区域的 Owner 为运维团队的所有成员。



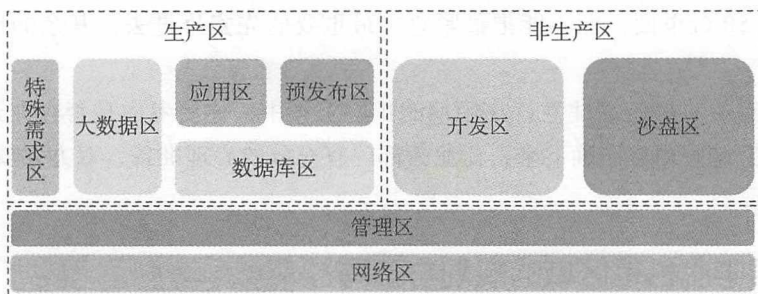
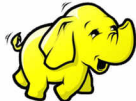


图 3-1 区域分类

### （3）数据库区

数据库区用于安置数据库服务器。当然，你可以根据实际情况再做细分。例如，根据数据库类型来划分（Oracle、MySQL 等），或者根据业务等级来划分（核心数据库、普通数据库等）。如果数据库需要外接存储，则需要综合考虑存储的空间占位与能耗，带存储的数据库最好和不带存储的数据库分开安置。该区域的 Owner 为运维 DBA。

### （4）应用区

应用区用于安置前端应用服务器。它通常位于网络拓扑的 DMZ 区。应用区是设备数量最多的区域，同时也是变数最多的区域。这需要你事先预留出充裕的空间。该区域的 Owner 为 PE（产品工程师 / 应用运维工程师）。

### （5）大数据区

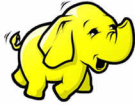
由于对计算能力和存储空间的要求很高，大数据服务器的能耗非常惊人。以 Intel E5-2640 v3 加 12 块 4TB 磁盘为例，350W 的整机能耗只能算正常值。如果计算任务比较繁重，则其峰值会更高。所以大数据服务器不可以和一般机型混放在一起。为了提升空间利用率，建议你为数据节点服务器选用电力容量更高的机柜。名称节点服务器等其他设备和普通服务器相比，则没有太大的差别。你可以考虑在大数据区内部划出一块地方，专门安置它们。

### （6）预发布区

预发布区属于准生产区域。研发和运维是两个体系，生产系统不允许研发人员直接操作。产品代码在完成测试之后，都是交付给运维团队负责上线的。测试环境和生产环境之间可能会存在一定的差异。如果上线发布后出现异常，则需要回退操作。代码有问题是很常见的，回退操作将严重影响系统的可用率。而预发布区可用于模拟线上的真实环境，进一步保证了生产系统的安全更新。

### （7）特殊需求区

特殊需求区主要用于承接各式各样的“非主流”需求。例如，构建特定的隔离环境、安置非标准配置的服务器、临时迁移或借调设备等，这些特殊情况都不适合做统一的安置。



此时，你可以把它们统统都放置在这里。我把它也戏称为“奇葩需求区”，意思是：不管你提出什么千奇百怪、偏离常态的需求，我这儿都可以满足。这个区域的空间应当多预留一些。如果用不完，后期可以慢慢回收。我的原则就是：你可以乱，但只能乱一点儿。一定要把维护成本限定在可控的范围之内。

特殊需求区的安置位置是有讲究的，建议放在应用区和数据库区的中间。另外，它的使用方式也有所不同，应当从中间的机柜向两侧扩展使用。因为要应对“需求气球”的变化，一开始会预留很大的空间，随着应用、数据库服务器的不断增加，这个空间会被逐步压缩。这种安置和上架的方式，体现了较强的灵活性，可以从容地应对未来有可能发生的需求数量的变化。

#### （8）开发区

开发区主要面向研发和测试人员。如何定义开发区的空间大小，这取决于团队规模和产品种类。理论上，开发区在业务扩张时期的需求量最大，但不会无休止地增长。由于开发产品的团队很多，为了防止干扰，开发区内部也存在着逻辑隔离的需求，我们通常管它叫闭环系统。

#### （9）沙盘区

沙盘区用于实验论证，它是为新技术探索研究或者故障复现而设立的。沙盘区不需要预留很多，一般不超过六个机柜。但它会带来比开发区更多的风险，所以沙盘区必须实施隔离。

### 3.2.3 规划布局案例

说了这么多，我们来举一个规划布局的实际案例。如图 3-2 所示，这是某个数据中心的机房平面局部图。

模块 3 中的核心设备即网络设备。部署网络设备应尽量靠近列头柜、配电柜等基础设施，还要注意对线缆长度的影响。穿线路由与线缆距离可通过施工平面图加以了解。

特殊需求区也位于模块 3，总计 20 个机柜。推荐从 R061 向上部署，一直部署到 R070，然后再从 R075 开始使用。如果将来机柜有富裕，可以根据实际情况分给应用区或者数据库区进行扩容。

应用区实际上是由两部分组成的，模块 1 的 R003~R010 和 R017~R024 是一组 PoD，模块 2 的 R031~R038 和 R045~R052 则是另外一组 PoD。在规划设计时，要注意 PoD 的限制，同一网段内的 IP 地址无法跨 PoD 分配。

和应用区相同，测试区也横跨了模块 1 和模块 2。它是由模块 1 的 R011~R014、R025~R028、以及模块 2 的 R039~R042、R053~R056 这四部分组成的。我们看到，测试区和应用区的两组 PoD 各是 16 个机柜，这三者构成了一个“品”字形的设计。假设上北下南，从北门进入模块后，左右两侧的 4 个机柜属于测试，剩下的就都是生产了。两个模块





|      |      |      |      |      |      |      |      |      |       |
|------|------|------|------|------|------|------|------|------|-------|
| 核心设备 | R014 | 模块 1 | R028 | 模块 2 | R042 | 模块 3 | R056 | 模块 4 | 测试区   |
|      | R013 |      | R027 |      | R041 |      | R055 |      | 管理区   |
|      | R012 |      | R026 |      | R040 |      | R054 |      | 应用区   |
|      | R011 |      | R025 |      | R039 |      | R053 |      | 数据库区  |
|      | R010 |      | R024 |      | R038 |      | R052 |      | 特殊需求区 |
|      | R009 |      | R023 |      | R037 |      | R051 |      | 沙盘区   |
|      | R008 |      | R022 |      | R036 |      | R050 |      |       |
|      | R007 |      | R021 |      | R035 |      | R049 |      |       |
|      | R006 |      | R020 |      | R034 |      | R048 |      |       |
|      | R005 |      | R019 |      | R033 |      | R047 |      |       |
|      | R004 |      | R018 |      | R032 |      | R046 |      |       |
|      | R003 |      | R017 |      | R031 |      | R045 |      |       |
| R002 | R016 | R030 | R044 |      |      |      |      |      |       |
| R001 | R015 | R029 | R043 | ODF  |      |      |      |      |       |
| 配电柜  | 配电柜  | ODF  | 配电柜  | 配电柜  | ODF  |      |      |      |       |
| 模块 1 |      | 模块 2 |      | 模块 3 |      | 模块 3 |      |      |       |
| 核心设备 | R070 | 模块 3 | R084 | 模块 3 |      | 模块 3 |      | 模块 3 |       |
|      | R069 |      | R086 |      | 配电柜  |      | 配电柜  |      |       |
|      | R068 |      | R082 |      | R096 |      | R108 |      |       |
|      | R067 |      | R081 |      | R095 |      | R107 |      |       |
|      | R066 |      | R080 |      | R094 |      | R106 |      |       |
|      | R065 |      | R079 |      | R093 |      | R105 |      |       |
|      | R064 |      | R078 |      | R092 |      | R104 |      |       |
|      | R063 |      | R077 |      | R091 |      | R103 |      |       |
|      | R062 |      | R076 |      | R090 |      | R102 |      |       |
|      | R061 |      | R075 |      | R089 |      | R101 |      |       |
|      | R060 |      | R074 |      | R088 |      | R100 |      |       |
|      | R059 |      | R073 |      | R087 |      | R099 |      |       |
| R058 | R072 | R086 | R098 |      |      |      |      |      |       |
| R057 | R071 | R085 | R097 | ODF  |      |      |      |      |       |
| 配电柜  | 配电柜  | ODF  | 配电柜  | 配电柜  | ODF  |      |      |      |       |

图 3-2 区域规划案例

ODF 全称叫作 Optical Distribution Frame，是专为光纤通信机房设计的光纤配线架设备。它会占用掉一部分空间，影响设备的上架率。一般来说，应用区的设备能耗低、上架密度高，所以应尽量绕过 ODF 部署。这里我们把含有 ODF 的机柜分配给了网络区、管理区和数据库区。相对的，这些区域对上架率不太敏感，可以忽略 ODF 带来的影响。

翻回头来，我们再来看一下模块 3。其实这里有一点点小问题：网络设备占据了六个机柜，但它采用了五五开的划分方式，使得 R060 和 R074 这两个机柜落了单。规划设计时最为忌讳的就是这种部署形式。但生米已经煮成熟饭了，于是我们让紧邻配线架的那个机柜落单，尽可能减少线缆的铺设长度。在这个案例中，规划不是同一个人做的。网络设备先



将位置给占了，等到规划服务器时，将 R060 和 R074 这组做成了沙盘区，因为沙盘区没有高可用的需求。如果沙盘区也要做双链路冗余，可以把两个 TOR 交换机都放置到一个机柜里面，另一个机柜要绕线的话，R060、R074 距离 ODF 和网络设备也是最近的。

### 3.3 规划设计心得

规划设计工作的重点在于：保证各个子系统之间的平衡，让各方的利益达到最大化。在这个过程中，难免会遇到一些左右为难的情况。本节将其中的一些典型案例以问答的形式列举出来，和读者朋友们一同分享。

#### 1. 数据中心机房的楼层应当如何选择

数据中心通常是多层建筑结构。如果是新建成的数据中心，客户较少，可供选择的余地较大。假设我们可以选择任意楼层的机房，那么是挑高层好，还是选低层更合适呢？

相对来说，高层的运输便利性差一些，等电梯比较浪费时间。但是低层也有低层的缺点。如果你需要使用时间源设备，就必须在楼顶加装卫星天线。低层的穿线路由距离长，将会增加施工的难度。比如信号的传输距离限制、布线空间要求等。

因此，数据中心低于三层的倾向于选低层的机房，反之就尽可能地挑高层使用。另外，请优先选择和办公区客户调试间同层的楼层。如果你在三楼驻场，而机房却安排在一楼，工作起来肯定很别扭。有些数据中心管理比较严格，对楼道施加门禁系统，只提供一部电梯给访客使用。不在同一楼层的话，工作效率和响应时间都会受到比较大的影响。如果你的设备保有量很大，机房分布在多个楼层内，核心层的网络设备适宜安置在中间楼层，便于平衡机房之间的穿线距离。

#### 2. 机柜不加装前门行吗

我们在一些新型的数据中心里可以看到，机柜是没有加装前门的。这和我们传统印象中的机柜不太一样。这种做法真的好吗？其实不装前门是有道理的。原因有三个。

第一，增加成本。传统机柜的布局没有模块概念，都是单摆浮搁的。而新型的模块式设计，会在两端部署门禁系统。不论制冷还是安全，都是以模块为单位考虑的。没有理由为单个机柜再次施加前门防护。

第二，影响制冷。在第2章中，我们已经讲过空调系统的送风模式了。现今主流的送风模式为下出风，冷空气要从机柜的前方进入。加装前门会影响气流和最终的制冷效果，浪费能源。

第三，操作不便。机柜后方只用于布线和上下架，日常操作都是在机柜前面完成的。机房内的操作空间本来就有限，来来往往经常有人员经过，这会儿你再开个门挡路就不太好了。

但有些时候，单个机柜实施全封闭是必需的。以金融业务为例，安全法案强制规定某些特殊设备必须实行物理隔离。此外，设备保有量小的用户，也会有同样的需求。

#### 列举一个案例：

假设我们有 500 台服务器，其中数据库 100 台，前端应用 370 台，管理设备 30 台。数据中心的一个模块里面可以安置 400 台服务器。由于资金有限，只购买了一个独立模块，剩下的 100 台服务器要和其他用户的设备混放在另外一个模块里面。应当如何安排？

考虑未来扩容的可能性，建议尽量使用独立的模块。我们可以和数据中心谈，空出来的机柜暂时预留下来。当然，这跟未来的保有量以及扩容的时间周期都有关系。如果一定要和别人混放，请注意以下两点。

第一，千万不要和竞争对手放在一块。按道理讲，莫说在同一个模块里，在同一个数据中心里面遇到竞争对手都不应该。

第二，不要在“混放区域”中安置核心设备。核心设备包括数据库和网络设备，你可以把一些测试机放进去。如果无法避免，比如我一共就 50 台服务器，而且以后也不打算增加投入。那不如就加装个前门吧，反正也没几个钱，安全第一。

### 3. 线缆标签如何管理

采用传统的本端到对端的标记形式过于繁杂。由于信息长度不一致，排版和打印起来都很麻烦。而且标签的制作和张贴滞后于设备上架，工作效率极低。如果我们事先定义好接线规则，完全可以使用类似 SN 的管理模式。给每一根线缆分配一个 SN，寻线时可以按照接线规则找到对端，只要线缆两端的 SN 是一致的即可。

### 4. 高电机柜好不好

高电机柜是互联网时代的产物。大型互联网公司的业务增长速度飞快，它们的设备规模已经非常巨大了。为了节约成本，对于机柜电力容量的需求正在逐步增加。传统数据中心的电力容量低，出现了设备上架率低、空间浪费大、工作效率差、管理成本高等一系列问题。高电机柜能够容纳更多的服务器，这有利于设备上架率的提升。数据中心每年是按照机柜数量来计费的，如果设备上架率低，无形中会增加很多成本。这对数据中心是有利的，但用户却吃了大亏。

可能有些人会担心高电机柜成本高，而且安置那么多设备，相当于在一个篮子里放了更多的鸡蛋。要是篮子打翻了，损失会不会更大呢？

如果体量够大，选用高电机柜还是很合算的。一个机柜每年的成本大约是十万元左右。如果你在数据中心拥有 1000 个机柜，即便能够节省 1%，那也是相当可观的了。而且规模上去以后，业务基本上都是分布式的，没有可能出现某个业务的设备都挤在同一个机柜里面。这种担心是没有必要的。

不过话说回来，电力容量并不是越高越好。电力容量会影响上架率，同时进一步影响

了网络设备的成本。这是一个多因素综合比较的过程。另外，高电机柜的资源非常有限，尤其是在一些二三线的城市就更加稀缺了。所以，也不必一味地求高，经济实用还是我们一贯的基本原则。

### 5. 一次性部署业务时有必要考虑位置冗余吗

既然前面讲到了业务的离散分布，那么假如一开始业务量少，只有几台服务器，我是不是还要考虑把它们分散到不同的机柜当中去呢？

我个人认为这种忧虑是多余的。说到这里，大家可能会有不同的意见了。你前面一再强调冗余的重要性，又讲部署规划时不能留有任何隐患，怎么现在又要全部推翻呢？

木桶原理告诉我们：一个木桶能装多少水，取决于它最短的那块木板。但这个被称之为“短板效应”的理论已经过时了。今天我们认为：一个木桶能装多少水，取决于它最长的那块木板（如图 3-3 所示）。也就是说，只要你在某一方面拥有足够的优势，就可以利用它去弥补自身的不足。

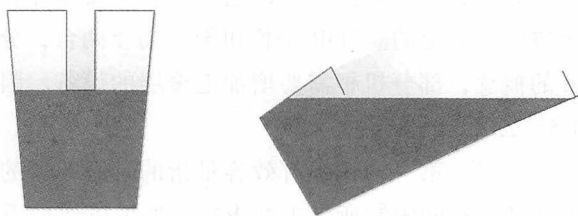


图 3-3 木桶原理的短板效应与长板效应

在今天，你认为机柜掉电的可能性会很高吗？如果你的答案是肯定的，那么分散到两个机柜后就没有问题了吗？要是按照这种逻辑思考，即便分散到两个机柜，整个数据中心不是也有掉电的可能吗？如此一来，难道你还要将它们分散到不同的数据中心去吗？

解决基础架构的问题，永远都是先从整体入手。如果我们把个体问题排在前面，那你的整体规划永远也实现不了。在小地方徘徊，会陷入一个无底洞。如果业务 A 要考虑，那么业务 B、业务 C 和业务 D 呢？这样一弄，自己就把自己给搞死了。

其实，一个机柜里最多不过二三十台设备。与其在细枝末节上浪费精力，还不如把整体的可靠性做扎实来得实在。如果电力系统能提供足够的保障，还有什么可纠结的呢？抓大放小，把关键点落实好，其他问题自然就迎刃而解了。在这一点上，思维模式必须要上升到一个更高的层级。随着体量的增加，所能实施冗余的手段也会越来越丰富。比如使用分布式的设计结构、多机房冗余等。完善整体架构才是消除个体问题最好的手段。

### 6. 成本核算时需要考虑什么

数据中心是一个持续性投入的项目，因此我们必须对采购成本加以控制。既然它是按照机柜数量来付费的，那我们是不是可以这样理解：用最少的机柜，上最多的设备。说白



了，就是提升上架率。当然，这不是绝对的。上架率并不是越高越好。如果一个机柜最多能容纳 25 台服务器，你真的会把它上满吗？我想没有人会这么做，因为根本就没有 25 口的交换机。

首先要明确设备的能耗值，这个值和服务器的配置有关。然后据此计算最佳的设备上架率，将上架设备的数量调整到一个适合的范围，使之和选用交换机的产品规格相适应。不要一边提升了上架率，另一边又增加了网络资源的开销。

在进行计算成本的时候，会涉及如下几组概念与公式。

### （1）机柜有效容量

这项参数直接反映了机柜的容纳能力，它的高低取决于电力容量、空间和设备能耗之间的匹配程度。计算机柜有效容量可以使用公式 3-1 来完成。

$$\text{机柜有效容量} = (\text{电力容量} - \text{常量值}) \div \text{设备最大能耗} \quad (3-1)$$

公式 1 中的常量值是指除服务器之外的设备能耗，比如 TOR 交换机。由于交换机的能耗比较低，一般在几十瓦到一百多瓦之间，所以它不像服务器那样有比较大的波动。我们可以近似地认为，这个数值是恒定的。TOR 交换机至少需要两台，分别用于业务和带外管理。此外，由于有 Pod 的概念，部分机柜需要增加汇聚层的设备。因此，我们建议将常量值的取值范围定义为 1.5~2A。

另外，还要特别注意几点。第一，机柜有效容量指的是服务器的数量，并不包含交换机。第二，这个公式无法消除空间限制所带来的影响。如果机架采用标准 U 位，设备是无法紧密摆放的。第三，使用最大能耗作除数更保险一些。这样做，是为了防止所有设备同时达到峰值而建立的最后一道防线。

### （2）空间利用率

空间利用率也称设备上架率，它是反映机柜容纳能力的另一种形态。计算空间利用率可以使用公式 3-2 来完成。

$$\text{空间利用率} = \text{设备占位数} \div \text{机柜 U 位总数} \times 100\% \quad (3-2)$$

由于服务器规格不同，单纯看机柜有效容量并不客观。同样都是 42U 的机柜，一个安置了 20 台 1U 的服务器，另一个安置了 15 台 2U 的服务器，你很难直接判断哪一个机柜利用得更合理。而空间利用率就很直观，参考意义更高。显然，第一个机柜的空间利用率不足 50%。我们认为，要是它能提升一些电力容量，肯定会获得更好的结果。

理论上，电力容量越高，可容纳的设备就越多。但机柜空间、设备规格、设备能耗以及网络拓扑都会制约机柜的有效容量。就像前面所举的那个例子，一个机柜最多能容纳 25 台服务器，如果上满了，空间利用率倒是上去了，但是交换机的资源又浪费了。所以电力容量不一定越高越好，需求要切合实际情况。

### （3）电量利用率

虽然空间利用率很直观，但它并不能和成本最优画等号。我们还要结合电量利用率来

进行综合评判。电量利用率反映了用电效率。换言之，就是看你有没有将电力资源充分地利用起来。计算电量利用率可以使用公式 3-3 来完成。

$$\text{电量利用率} = (\text{设备数量} \times \text{设备日常能耗} + \text{常量值}) \div \text{电力容量} \times 100\% \quad (3-3)$$

设备的能耗值有两个：一个是日常能耗，另一个业务繁忙时的最大能耗。注意：在计算电量利用率时，应当使用设备的日常能耗值，而不是最大能耗值，因为常态才是最真实的。

#### (4) 单台损失

由于我们不可能将电力资源全部耗尽，所以电量利用率肯定小于 1。如果我们将无法使用的电量算作亏损，单台损失就是平摊这部分亏损后的价格。计算单台损失可以使用公式 3-4 来完成。

$$\text{单台损失} = \text{机柜费用} \times (1 - \text{电量利用率}) \div \text{机柜有效容量} \quad (3-4)$$

通过公式 3-4 我们能看到，电量利用率越高，损失越小，空间利用率越高，平摊下来的成本就越低。请注意，公式的分母使用了机柜有效容量，没有包含交换机。这是因为交换机的个数是一个定值。不管怎样，只要启用一个机柜，交换机的数量都是固定的。而且交换机能耗低，对电量利用率的影响也不大。综合这两点来考虑，它不适合做平摊的对象。

### 7. 如何进行成本核算

我们举个实例。假设现在有两家数据中心。数据中心 A 提供 16A 的机柜，单价费用为 5000 元。数据中心 B 提供 40A 的机柜，单价费用为 12 800 元。机柜规格均为 42U 位的非标准机柜。服务器日常能耗值为 0.8~1A，最大能耗值为 1.2A，常量值设定为 1.5A。服务器的规格为 1U，保有量 10 000 台。

第一轮，我们先看单价费用除以电力容量。

$$\text{数据中心 A 的电力单价} = 5000 \div 16 = 312.5 \text{ (元/A)}$$

$$\text{数据中心 B 的电力单价} = 12\,800 \div 40 = 320 \text{ (元/A)}$$

这一轮是数据中心 A 赢了。那么接下来，我们使用公式 1 来计算机柜的容纳能力。

$$\text{机柜有效容量} = (\text{电力容量} - \text{常量值}) \div \text{设备最大能耗}$$

$$\text{数据中心 A 的机柜有效容量} = (16 - 1.5) \div 1.2 = 12 \text{ (台)}$$

$$\text{数据中心 B 的机柜有效容量} = (40 - 1.5) \div 1.2 = 32 \text{ (台)}$$

这次数据中心 B 甩了它的竞争对手好几条街。第三轮，我们使用公式 2 来计算机柜的空间利用率。

$$\text{空间利用率} = \text{设备占位数} \div \text{机柜总位数} \times 100\%$$

$$\text{数据中心 A 的空间利用率} = 12 \div 42 \times 100\% = 28.57\%$$

$$\text{数据中心 B 的空间利用率} = 32 \div 42 \times 100\% = 76.19\%$$

在空间利用率上，数据中心 B 依旧优于它的对手。我们再利用公式 3 做个对比，看哪一家的电量利用率最好。

电量利用率 = (设备数量 × 设备日常能耗 + 常量值) ÷ 电力容量 × 100%

数据中心 A 的电量利用率 =  $(12 \times 1 + 1.5) \div 16 \times 100\% = 84.37\%$

数据中心 B 的电量利用率 =  $(32 \times 1 + 1.5) \div 40 \times 100\% = 83.75\%$

这次，数据中心 A 又扳回一局。最后，我们借助公式 4 检视一下单台损失的情况。

单台损失 = 机柜费用 × (1 - 电量利用率) ÷ 机柜有效设备数

数据中心 A 的单台损失 =  $5000 \times (1 - 0.8437) \div 12 = 65.125$  (元)

数据中心 B 的单台损失 =  $12\,800 \times (1 - 0.8375) \div 32 = 65$  (元)

按照一万台服务器的保有量计算，采购数据中心 A 需要 834 个机柜，而采购数据中心 B 则只需要 313 个机柜。服务器的机柜采购成本如下所示。

$834 \times 5000 = 417$  (万元)

$313 \times 12\,800 = 400.64$  (万元)

如表 3-1 所示，这是两家数据中心（服务器）机柜成本的对比结果。

表 3-1 两家数据中心的数据汇总比较

| 参 数             | 数据中心 A | 数据中心 B |
|-----------------|--------|--------|
| 电力容量 (A)        | 16     | 40     |
| 机柜月租单价 (元)      | 5000   | 12 800 |
| 电力单价 (元 /A)     | 312.5  | 320    |
| 机柜有效容量 (台)      | 12     | 32     |
| 空间利用率 (%)       | 28.57  | 76.19  |
| 电量利用率 (%)       | 84.37  | 83.75  |
| 单台损失 (元)        | 65.125 | 65     |
| 服务器的机柜数量 (个)    | 834    | 313    |
| 服务器的机柜采购成本 (万元) | 417    | 400.64 |

注意，到此为止，这些都没有包含网络设备的成本。空间利用率还会直接影响网络设备的数量。如图 3-4 所示，由于业务网络需要采取冗余模式，我们将机柜两两分为一组。业务交换机要连接组内的所有设备，而带外管理交换机则只需连接本机柜的设备。采购数据中心 A 需要业务和带外管理交换机各 834 台，而采购数据中心 B 业务交换机的数量是有变化的。因为两个机柜的服务器总数为 64 台，一个 48 口交换机是无法支撑的。加入网络设备的费用后，TCO 反而有可能会大于数据中心 A，这就需要我们去做进一步的分析。

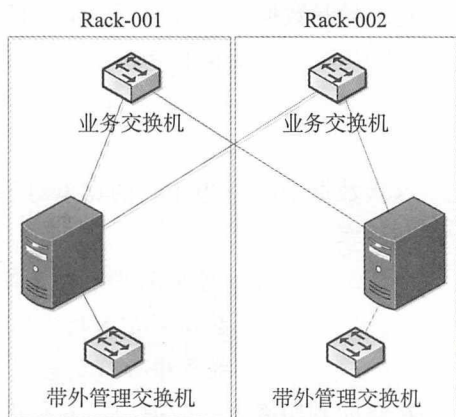


图 3-4 业务网络和带外管理网络的接线方式



我们从这个示例中获得的启示是：在进行成本核算时，不能孤立地看问题，要综合考虑各方面因素所带来的影响。事实上，管理成本和资源损耗的权重应当大于采购成本才对。随着时间进程的前行，采购成本会被平摊，但管理成本和资源损耗是持续输出的。管理成本 > 持续成本 > 一次性成本，这个思维是实现成本控制的关键。

### 3.4 本章小结

本章论述了数据中心规划设计的一些要点。好的方案才能有好的开局，方案设计也许是由一两个人完成的，但它最终会影响到成百上千的用户。

在设计规划中，不但要小心需求变化带来的各种麻烦，还要注意保持各个子系统之间的平衡性。规划设计始终贯穿在整个项目建设的进程之中，应当做好充分的调研工作，再进行缜密的部署安排。解决问题要从全局的角度入手，但也不应有细节上的疏忽和遗漏。

关于细节的重要性，我们将在下一章中为你讲述，网络规划细节是如何对系统运维产生影响的。



## 第4章

# 网络规划细节对系统运维的影响

俗话说“隔行如隔山”，专业的事情应当交给专业的人来做。不论是架构设计，还是资源规划，有经验的架构师能够很好地把控整体节奏。平衡性是方案设计当中的重、难点，如果设计者在制定方案时存在个人倾向，对一些细节问题考虑不周，可能会导致设计方案的失衡。就像下面的这个故事一样，设计者在为网络团队提供便利的同时，却给系统团队增添了无数的烦恼。

### 运维故事 8：一次存储迁移引发的思考

“什么？到现在还没有开始？”

陆风抬手看看手表，现在已经是下午三点半了。区区两套存储，从上午十点到现在还没有完成迁移，真的有那么难吗？

没办法，他只好匆匆结束本来正在进行的工作会议，只身打车赶到位于本市近郊的一家数据中心。陆风真是受够了人口集中城市化所带来的种种烦恼。他本来心里就很着急，却偏偏又赶上了堵车。也难怪，这条路的交通拥堵在全市可都是出了名的。

所以，等他赶到数据中心时，已经是下午五点半了。推开门一看，一帮人正大眼瞪小眼、像盼着救星似地期待着他的出现。“哥，你终于来了！”首先迎上来的就是阿祥的一张笑脸。这工作本来是由他负责的，不用问，一定是搞砸了。

“设备我们已经下架了，现在就是有点儿小问题。”阿祥陪着笑脸继续说道，“本来是想一台一台搬的，但是他们（指 IDC 服务团队）说一块儿搬多快啊，一台一台太麻烦。”

一块儿搬的？恐怕坏事就坏在这“一块儿搬”上面了。陆风意识到，阿祥这小子所说的小问题，显然并不那么简单。

“然后呢？”

“然后我们就把光纤都拆了，设备也上架了，但是现在有点儿弄不清楚了。”

弄不清楚？这句话宛如一声炸雷在陆风的大脑里响起，难道说……

“小问题是，我们还修改了一部分迁移的新地址。所以呢，我现在觉得有点儿乱，不知



道线缆该怎么连回去了。”

陆风现在只觉得头一阵阵地眩晕。他心里搞不明白，老严为什么要把这样一个人交给他来带？阿祥并不是一个新人，只是以前不是做这个方向的。然而，他做事毛躁、粗心大意的毛病陆风可是早有耳闻。听人说，很多团队都因为这个不愿和他一起合作，老严却偏偏把他塞给了自己。老严觉得，阿祥在这方面虽然没什么经验，但还是有一定能力的，多带带就好了。除此之外，他还有一个很充分的理由就是——陆风这组人手少。陆风是个谨慎的人，这次要不是真的忙不过来，他是不会让阿祥去的。为了避免出问题，陆风还特意做好了前置工作，发邮件给阿祥，千叮咛万嘱咐地告诉他所有的操作流程和注意事项，没想到还是出事儿了。

“实施之前我是怎么嘱咐你的？不是说让你先记录好对应关系后，改好一台迁移一台么？”

“哦，我给忘啦，呵呵。”

这下事情就变得非常麻烦了。配置存储时会有一张对应关系表，是服务器的HBA卡与光纤交换机端口的映射关系。一般情况下，会用业务地址和光纤交换机的端口做对应。但是在机房现场，不登录系统是看不到业务地址的，只能通过服务器的液晶显示屏看到带外管理地址。而这两套地址之间偏偏没有清晰明了的对应关系。

陆风一抬头，正看见存储工程师老张站在一边，他急忙把老张拉了过来，小声问道：“你还记得原来的顺序吗？”

“我哪里记得什么带外管理地址，原来的连线规则我倒是清楚，可现在业务地址和设备都给他们弄乱了，我也搞不清楚啊。”

说完老张又叹口气，“唉，别提了。我上午在忙另外一个项目，下午三点多才赶到现场。结果一来就发现成这样了，我当时要是在场，还能出这事儿？本来个把小时就能搞定的，现在不知道要弄到什么时候呢？我明天还得出差呢，这一周都回不来。这个活儿要是今天干不完，那就只能下周见了。”

哪儿能拖到下周？业务部门急等着用呢！迁移工作是今天必须完成的任务。但问题是，阿祥拆了线缆却没做记录，还弄乱了原有的地址映射关系，IDC的人也不敢动，陆风只能靠自己了。好在他有原始记录，找回对应关系倒也不难。但迁移工作还涉及对地址的修改，阿祥只改了一部分，也就是说，这里面既有新地址，也有老地址。现在他已经无法相信阿祥了，只能根据原始记录一一核对。但是，这两套地址既不连续，又没有清晰的对应关系，谁也不挨着谁。在这样一个乱糟糟的环境下，要确保顺利完成迁移谈何容易？陆风做了两个深呼吸，试着让自己冷静下来。他决定先把所有的旧地址都改过来，顺便确认一下SN号，免得那小子糊里糊涂地改错了机器。然后做一张新旧地址对照表给老张，安排他调整存储的配置。最后再统一检查一遍。

光修改地址和确认搬迁，陆风就花了两个多小时。等他和老张忙完走出机房的时候，



已经是深夜十点多钟了。俩人连晚饭都没有吃，老张拖着疲惫的身子回去了，而陆风今天在公司未完成的工作还要明天加班才能赶上。

## 4.1 案例复盘

好了，故事讲完了。我们对这次事件做个复盘，看看陆风和老张为什么如此痛苦？

创业之初，只有十几个人，七八条枪。哦，不对，是七八台服务器。设备数量少，业务也很清晰。网络地址的分配是按着顺序使用的，如表 4-1 所示。

表 4-1 业务地址与业务的对应关系表

| 业务地址       | 业务类型 | 业务地址     | 业务类型  |
|------------|------|----------|-------|
| 10.0.100.1 | 应用 A | 10.0.0.1 | 数据库 A |
| 10.0.100.2 | 应用 A | 10.0.0.2 | 数据库 A |
| 10.0.100.3 | 应用 B | 10.0.0.3 | 数据库 B |
| 10.0.100.4 | 应用 B | 10.0.0.4 | 数据库 B |

但是随着业务的扩展，应用种类和设备数量都在飞速增长，如表 4-2 所示。慢慢地，网络工程师老刘发现调整防火墙策略是越来越费劲了。因为安全策略要求：业务 A 访问数据库 A，业务 B 访问数据库 B，相互不能交叉。自从防火墙策略采取了点到点的方式以来，老刘每天都有做不完的工单，他感觉自己都要“死”在岗位上了。

表 4-2 扩容后业务地址与业务的对应关系表

| 业务地址        | 业务类型 | 业务地址     | 业务类型  |
|-------------|------|----------|-------|
| 10.0.100.1  | 应用 A | 10.0.0.1 | 数据库 A |
| 10.0.100.2  | 应用 A | 10.0.0.2 | 数据库 A |
| 10.0.100.3  | 应用 B | 10.0.0.3 | 数据库 B |
| 10.0.100.4  | 应用 B | 10.0.0.4 | 数据库 B |
| 10.0.100.5  | 应用 A | 10.0.0.5 | 数据库 A |
| 10.0.100.6  | 应用 A | 10.0.0.6 | 数据库 B |
| 10.0.100.7  | 应用 A |          |       |
| 10.0.100.8  | 应用 B |          |       |
| 10.0.100.9  | 应用 B |          |       |
| 10.0.100.10 | 应用 A |          |       |

老刘想了一个好办法。他根据业务划分了若干子网，将不同的数据库归属到不同的子网当中，防火墙策略只需要配置一次就行了。地址规划自然也是老刘分内的事情。他可是个过日子的仔细人，坚决不肯浪费地址资源。这不，新的业务需求来了。数据库需要扩



容 190 台服务器，但第一批到货的设备只有 20 台服务器和两套存储。于是，他采用了如表 4-3 所示的方式来规划。

表 4-3 数据库扩容需求地址资源规划表

| 业务类型  | 计划数 | 地址池           | 可用地址数 | 子网 | 第一批到货数 |
|-------|-----|---------------|-------|----|--------|
| 数据库 A | 30  | 10.0.0.0/27   | 30    | A  | 2      |
| 数据库 B | 50  | 10.0.0.64/26  | 62    | B  | 6      |
| 数据库 C | 70  | 10.0.0.128/25 | 126   | C  | 8      |
| 数据库 D | 10  | 10.0.1.0/28   | 14    | D  | 2      |
| 数据库 E | 30  | 10.0.0.32/27  | 30    | E  | 2      |

网络地址分为业务地址和带外管理地址。表 4-3 所展示的内容，是老刘针对业务地址的规划情况。至于带外管理地址的分配，则依旧延续以前的风格，按着顺序一个一个地使用。具体到本次实施的这个项目，设备地址的详细对应关系如表 4-4 所示。

表 4-4 第一批设备地址对应关系表

| 业务地址       | 带外管理地址      | 业务类型  | 子网 |
|------------|-------------|-------|----|
| 10.0.0.1   | 172.16.0.1  | 数据库 A | A  |
| 10.0.0.2   | 172.16.0.2  | 数据库 A | A  |
| 10.0.0.65  | 172.16.0.3  | 数据库 B | B  |
| 10.0.0.66  | 172.16.0.4  | 数据库 B | B  |
| 10.0.0.67  | 172.16.0.5  | 数据库 B | B  |
| 10.0.0.68  | 172.16.0.6  | 数据库 B | B  |
| 10.0.0.69  | 172.16.0.7  | 数据库 B | B  |
| 10.0.0.70  | 172.16.0.8  | 数据库 B | B  |
| 10.0.0.129 | 172.16.0.9  | 数据库 C | C  |
| 10.0.0.130 | 172.16.0.10 | 数据库 C | C  |
| 10.0.0.131 | 172.16.0.11 | 数据库 C | C  |
| 10.0.0.132 | 172.16.0.12 | 数据库 C | C  |
| 10.0.0.133 | 172.16.0.13 | 数据库 C | C  |
| 10.0.0.134 | 172.16.0.14 | 数据库 C | C  |
| 10.0.0.135 | 172.16.0.15 | 数据库 C | C  |
| 10.0.0.136 | 172.16.0.16 | 数据库 C | C  |
| 10.0.1.1   | 172.16.0.17 | 数据库 D | D  |
| 10.0.1.2   | 172.16.0.18 | 数据库 D | D  |
| 10.0.0.33  | 172.16.0.19 | 数据库 E | E  |
| 10.0.0.34  | 172.16.0.20 | 数据库 E | E  |

因为数据中心属于第三方托管，陆风需要指导 IDC 服务团队的上架工作。为了便于安排施工，他根据老刘的分配方案重新设计了表格，如表 4-5 所示。

表 4-5 设备上架及地址对应关系表

| 1 <sup>st</sup> Gourp |             |           | 2 <sup>nd</sup> Gourp |             |           |
|-----------------------|-------------|-----------|-----------------------|-------------|-----------|
| 业 务 地 址               | 带外管理地址      | 设备上架顺序    | 业 务 地 址               | 带外管理地址      | 设备上架顺序    |
| *                     | *           | 光纤交换机 A-1 | *                     | *           | 光纤交换机 B-1 |
| *                     | *           | 光纤交换机 A-2 | *                     | *           | 光纤交换机 B-2 |
| *                     | *           | 存储 A      | *                     | *           | 存储 B      |
| 10.0.0.1              | 172.16.0.1  | 数据库 A     | 10.0.0.2              | 172.16.0.2  | 数据库 A     |
| 10.0.0.65             | 172.16.0.3  | 数据库 B     | 10.0.0.66             | 172.16.0.4  | 数据库 B     |
| 10.0.0.67             | 172.16.0.5  | 数据库 B     | 10.0.0.68             | 172.16.0.6  | 数据库 B     |
| 10.0.0.69             | 172.16.0.7  | 数据库 B     | 10.0.0.70             | 172.16.0.8  | 数据库 B     |
| 10.0.0.129            | 172.16.0.9  | 数据库 C     | 10.0.0.130            | 172.16.0.10 | 数据库 C     |
| 10.0.0.131            | 172.16.0.11 | 数据库 C     | 10.0.0.132            | 172.16.0.12 | 数据库 C     |
| 10.0.0.133            | 172.16.0.13 | 数据库 C     | 10.0.0.134            | 172.16.0.14 | 数据库 C     |
| 10.0.0.135            | 172.16.0.15 | 数据库 C     | 10.0.0.136            | 172.16.0.16 | 数据库 C     |
| 10.0.1.1              | 172.16.0.17 | 数据库 D     | 10.0.1.2              | 172.16.0.18 | 数据库 D     |
| 10.0.0.33             | 172.16.0.19 | 数据库 E     | 10.0.0.34             | 172.16.0.20 | 数据库 E     |

但是，这张表中含有敏感的业务地址。为人谨慎的陆风是不可能将它直接交付给 IDC 服务团队的，他在提交表格时把业务地址的部分给删掉了。所以，IDC 服务团队只能看到带外管理地址和上架的顺序。请注意，存储设备和光纤交换机也是有管理地址的，但因为这些内容和本次复盘没有关系，我们暂且不提。

在配置存储的过程中，有一项参数名为 Hosts。为了便于管理，老张用业务地址来定义 Hosts 的名称。配置工作完成后，他给陆风提供了一张 Hosts-Port 对应关系表，如表 4-6 所示。

表 4-6 Hosts-Port 对应关系表

| 1 <sup>st</sup> Gourp |            |      | 2 <sup>nd</sup> Gourp |            |      |
|-----------------------|------------|------|-----------------------|------------|------|
| 带外管理地址                | Host       | Port | 带外管理地址                | Host       | Port |
| 172.16.0.1            | 10.0.0.1   | 1    | 172.16.0.2            | 10.0.0.2   | 1    |
| 172.16.0.3            | 10.0.0.65  | 2    | 172.16.0.4            | 10.0.0.66  | 2    |
| 172.16.0.5            | 10.0.0.67  | 3    | 172.16.0.6            | 10.0.0.68  | 3    |
| 172.16.0.7            | 10.0.0.69  | 4    | 172.16.0.8            | 10.0.0.70  | 4    |
| 172.16.0.9            | 10.0.0.129 | 5    | 172.16.0.10           | 10.0.0.130 | 5    |





(续)

| 1 <sup>st</sup> Gourp |            |      | 2 <sup>nd</sup> Gourp |            |      |
|-----------------------|------------|------|-----------------------|------------|------|
| 带外管理地址                | Host       | Port | 带外管理地址                | Host       | Port |
| 172.16.0.11           | 10.0.0.131 | 6    | 172.16.0.12           | 10.0.0.132 | 6    |
| 172.16.0.13           | 10.0.0.133 | 7    | 172.16.0.14           | 10.0.0.134 | 7    |
| 172.16.0.15           | 10.0.0.135 | 8    | 172.16.0.16           | 10.0.0.136 | 8    |
| 172.16.0.17           | 10.0.1.1   | 9    | 172.16.0.18           | 10.0.1.2   | 9    |
| 172.16.0.19           | 10.0.0.33  | 10   | 172.16.0.20           | 10.0.0.34  | 10   |

注意，带外管理地址这一栏是陆风自己加上去的。老张并不关心带外管理地址，但 IDC 服务团队在布线的时候需要它，而陆风又不能让他们看到 Hosts，所以在交付表格的时候，还要删掉 Hosts 的内容。

#### (1) 烦恼之一——子网掩码的转换

子网掩码的表达式有两种，一种是 Prefix-Length，另一种是点分十进制。在网络规划中通常会采用前者，因为它表述起来简明扼要。但是，IDC 服务团队的技术水平参差不齐，很多人看不懂 Prefix-Length，他们希望陆风能提供点分十进制的子网掩码。而老刘把地址段切分得太碎了，各种长度的掩码都有，做转换的时候非常麻烦。



#### 点评：

划分子网确实是节约资源的一种方法，但没有必要切分得太细。比方说，子网 D 的规划就是值得商榷的。老刘只顾眼前，却没有考虑到今后业务增长的可能性。

#### (2) 烦恼之二——存储工程师工作不开心

因为 Hosts 和端口号对应不起来。1 对 1、2 对 2 的不好吗？现在这种一团糟的对应关系，既难看又难记，这让老张觉得很很不爽。



#### 点评：

带存储的数据库的成本非常高昂。再怎么增长，也难以形成很大的规模。应当从实施和管理的角度上考虑，为此类业务提供一些必要的便利条件。牺牲一些地址空间，却获得了管理上的方便，这样做是值得的。我们不应以增加管理成本为代价，来换取有限的资源节约。

#### (3) 烦恼之三——两套地址难以对应

如果 IDC 服务团队在巡检时发现了设备故障，他们会通知陆风，并告知设备的带外管理地址。陆风接到告警后，要通知负责业务运维的小青。但是他没法直接转发告警消息，因为小青所关心的是业务地址，而非带外管理地址。

同样，当小青向陆风求助的时候，只会告诉陆风业务地址是多少。如果需要 IDC 服务

团队的现场协助，陆风还要提供对应的带外管理地址。

不管怎样，陆风在中间永远要充当翻译的工作。由于两套地址之间没有形成清晰明了的对应关系，陆风每次只能打开 Excel 表格做查询，真是麻烦死了。



#### 点评：

所有信息应当保存在 CMDB 中，负责地址转换的工作应当由系统自动完成。由于前期可能没有完善的系统，我们强烈建议，在两套地址之间建立一个易于理解的对应规则。

#### （4）烦恼之四——就是这次搬迁

表 4-6 中所示的 Hosts-Port 对应关系表是陆风整理完的。但老张和 IDC 服务团队所看到的内容，都只是这张表的一部分，而且他们的视图是不同的。由于 IDC 不愿为连线承担风险，所以才要阿祥协助现场指导。但是阿祥忘了陆风的叮嘱，一口气把线全给拆了。更要命的是，他还改了一部分新地址，把原有的对应关系给弄乱了，导致 IDC 的人也不敢动。老张觉得，要处理现在这种局面，非得让陆风亲自出马了。



#### 点评：

规划自有规划的问题，但在这里，我觉得责任心和工作态度是导致问题的直接因素。用一个不靠谱的人，没有什么比这更糟糕的了。

## 4.2 事情为什么弄得一团糟

这个案例中存在着如下几个问题。

第一，网络规划没有考虑到未来的业务增长，子网规模的类型过多，分配和维护工作的难度很大。

第二，每套存储对应了 10 个数据库服务器，而它们的业务地址是离散的，使得 Hosts-Port 的对应关系变得异常复杂。在后期的维护过程中，要管理好这些信息是相当吃力的。

第三，信息没有（或者是没有条件）存储在 CMDB 中，两套地址的转译工作要依靠人工查询来完成，效率极其低下。

那么，这些问题能不能改进呢？我认为答案是肯定的。

#### （1）减少子网的规模类型

之前我们讲到，随着业务的增多，点对点的防火墙策略让老刘吃不消了。这也就是为什么他要按照业务来划分子网的根本原因。但是，老刘没有考虑未来业务增长的可能性，把子网切得太碎了。这样做并不能节省很多资源，反而会给自己今后的管理工作带来麻烦。具体问题我们会在 4.3 节中进行分析。

#### （2）数据库地址分配的改进

那么，老刘这样做的动机是什么呢？我想原因无非有两个：第一，地址资源的利用率

最高；第二，他将来开策略比较方便。因为对他来说，同一个业务的地址是连续的。

老刘期望地址连续的想法很好，但这不太现实。随着扩容变化，新的地址段就会加入进来。所以，一个业务的资源池里面，地址不可能永远都是连续的。而且这种地址分配，给陆风和老张添了很多麻烦。一套存储里面不可能只部署一种业务，这种设计会导致一套存储出现多个不连续的地址。

这种做法，对 NE、DBA 和 SE 又会产生什么影响呢？表 4-7 中的内容，反映了三种角色对信息的需求比较。

表 4-7 信息需求比较

| 信息需求 | NE         | DBA     | SE                   |
|------|------------|---------|----------------------|
| 信息字段 | 防火墙策略、业务地址 | 业务、业务地址 | 业务地址、带外地址、Hosts、Port |
| 信息修改 | 增加、删除      | 增加、删除   | 增加、删除、变更             |
| 信息查询 | 单一关系       | 单一关系    | 多重关系                 |
| 维护频率 | 低          | 中       | 中                    |
| 操作难度 | 低          | 低       | 高                    |

首先，NE 和 DBA 都不需要登录存储设备和带外管理。因此，他们所查询的信息是一对一的关系。而 SE 在日常管理的操作中，会涉及多个信息字段，信息查询是多重对应的关系。

其次，防火墙策略是按区域设置的，除非有新的地址增加或者需求调整，定义后再次发生变动的概率很低。老刘给自己开了方便之门，但他也仅用了一次而已。而陆风要经常和这张信息表打交道，操作频率和复杂度要远远大于老刘。

简而言之，老刘希望自己管理的地址是连续的，于是把局部方案调整成对自己有利的形式。另一边，却把陆风管理的地址搞得七零八落，降低了整体的工作效率。

既然陆风的操作最多最复杂，如果他在维护时可以不去做查询，整体效率就会提升很多。地址连续便于管理，但只适合局部空间，所以这个方便应当留给最需要的人。这个案例中，数据库没有涉及 VLAN 的使用，而大多数防火墙也支持用地址集来定义对象。既然如此，一套存储对应十台服务器，地址结尾 1~10 是第一套存储，11~20 是第二套存储，是不是更好一些呢？

### （3）两套地址对应关系的改进

理想情况下，基础信息的获取应当来自于 CMDB，相关内容的推送也应由系统自动完成。但是一开始，完善的系统可能根本就不存在。如果必须执行人肉查询，还要设置各种障碍，那真是太糟了！在这种情况下，我们强烈建议设计一些简单明了的规则，尽可能地减少人工查询和大脑记忆。

好的方案是：让业务地址和带外管理地址之间的对应关系清晰化，实现相互间的直接转译。例如，表 4-8 中所示的就是一个示例。业务地址的第二段数值加 128，即为带外管理地址。



表 4-8 地址对应关系表示例

| 业务地址      | 带外管理地址      |
|-----------|-------------|
| 10.0.0.1  | 10.128.0.1  |
| 10.2.16.1 | 10.130.16.1 |

读者朋友们在这里可能会产生一个疑问。由于两套地址是完全对照的，所以它们的空间大小也是相同的，相当于资源消耗翻倍，这样做会不会太奢侈了？

我个人认为这是值得的。首先，如果生产环境有条件使用 A 类私有地址，可用资源就不是问题。其次，资源的适度损耗是必然的，也是合理的。别忘了，管理成本可是持续输出的。如果能让逻辑变得更加清晰、降低管理成本，这种做法就是“物有所值”的。

当然，能用系统自动化处理是最好的方案。但开发需要一个漫长的过程，在这个 Cold Down 时间结束前，地址直译的规则可能是最好的解决方案了。无须信息查询，你就可以用其中一套地址推导出另外一套来。另外需要注意的是，CMDB 只是一个数据库，即便它的内容是完整而权威的，也不等同于实现了信息自动化。如果信息查询工作还是基于手工操作，那么 CMDB 和 Excel 表没有任何的区别。我个人认为，实现关联消息的自动推送与填充，才是真正意义上的信息自动化。例如，监控系统在告警时，应当提供如下相关信息。

- ❑ 远程维护类信息：业务地址、带外地址等。
- ❑ 业务影响类信息：业务名称、部门、联系人等。
- ❑ 硬件报修类信息：IDC 及其联系人、机架位置、厂家、SN 等。

关于这些内容，我们会在后面的章节再做详细论述。

### 4.3 网络空间资源的规划

说起网络空间资源的规划，并不像我们想象中的那么简单。你可能会这样认为，不就是对照业务类型划几个子网么，有什么难的？事实上，这项工作涉及很多的细节。比方说，设计一个子网的空间，空间过大会形成资源浪费，但反过来又会增加管理成本，影响后续的扩容计划。在设计过程中，你要有相关的参照和依据来支撑你的方案。

#### 4.3.1 PoD 容量的计算方法

经典的三层网络架构是由核心、汇聚和接入三部分组成的。为了保障服务的伸缩性、灵活性以及维护工作的一致性，数据中心在建设过程中，采取了模块化的网络部署方式。它将汇聚层、接入层连同服务器等设备统统部署到同一个逻辑单元之中，我们将其称之为一个 PoD（Point of Delivery）。所有的 PoD 通过核心层设备形成互联。当数据中心需要扩

容时，只需水平增加 PoD 即可。

那么一个 Pod 究竟能够容纳多少台服务器和机柜呢？这个问题和机柜的规划布局有着非常紧密的联系。下面，我就举一个示例来说明。

假设，我们使用千兆以太网网络，每个机柜可安置 20 台服务器，每台服务器有两块千兆网卡，采用全负载模式。按照双机柜分组的形式计算，每组机柜的节点总数共计 40 个，总带宽为  $2\text{Gib} \times 40 = 80\text{Gib}$ 。如果接入设备有四个万兆的上联端口，则它的处理能力是  $40\text{Gib}$ 。一个 48 口的汇聚设备能够提供  $480\text{Gib}$  的总带宽，两台就是  $960\text{Gib}$ 。

$960$  除以  $40$  等于  $24$ 。也就是说，理论上可以连  $24$  台接入设备。但是，汇聚层设备还要上联到核心层，所以不可能把端口都用掉。又因为双机柜分组的缘故，每组的总带宽为  $80\text{Gib}$ ，接入设备上联带宽只有  $40\text{Gib}$ ，所以需要  $2$  台接入设备来支撑。如果按照一个 PoD 容纳  $20$  个机柜来折算，节点总数是  $400$  个，总带宽的需求就是  $800\text{Gib}$ 。

我们从这个示例中可以看出，机柜有效容量、带宽需求以及网络设备的处理能力都在影响着 PoD 的容量。这里所举的例子，已经是比较极端的情况了。通常来说，一个 PoD 内的物理节点总数不会大于  $400$  个。而子网无法跨越 PoD，你不可能在不同的三层上看到同一个子网。不然，路由设备在传递数据包时，根本分不清到底该往哪里送。所以我需要提醒大家一点的是，尽管我们强调不要把子网切得太碎，但凡事过犹不及，把子网划得太大也是错误的。因为，在一个 PoD 里浪费的地址无法在其他 PoD 中使用。

### 4.3.2 地址空间的规划

提前明确需求是地址空间规划的理想条件。但事实上，往往是计划赶不上变化。如果对情况预估不足，很容易出现问题。过度分配会造成资源浪费，可是抠得太紧又会影响扩容。

举个例子，假定一个 PoD 里最多能容纳  $400$  个节点。最初的业务规划是全部使用物理机，并且同属一个 VLAN。于是，我们分配了子网  $10.0.0.0/23$ 。但随后出现了需求变更，由于对业务量预估不足，现在需抽调  $150$  台物理机，将其转化为宿主机。每台宿主机部署  $8$  台虚拟机，地址总数增加到了  $1450$  个，也就是  $6$  个 C。如果将子网扩充到  $10.0.0.0/21$ ，之前已经部署好的系统只要改一下网关就行了。但不幸的是，子网  $10.0.2.0/24 \sim 10.0.7.0/24$  已经分给其他业务使用了，我们只能从  $10.0.8.0$  开始新的分配。如此一来，这个 VLAN 里等于包含了两个子网。两个子网在 VLAN 内是直接通信的，外出时则各走各的网关，相当于在路由上配置了多个子接口。

再举一个资源缩水的例子。最初的业务规划是全部使用虚拟机  $3200$  台，并且同属一个 VLAN，需要一个  $16\text{C}$  的超大子网  $10.0.0.0/20$ 。但是，部分业务在上线之后出现了虚拟机性能不足的问题，准备调整  $400$  个节点做物理机。服务器资源是新增的，下线的虚拟机作为资源池保留。此时，原有的 PoD 无法继续增加节点数量，新的服务器只能部署在另一个

PoD 下面。尽管是同样的业务，但原有子网的空闲地址都无法使用了，这将造成严重的资源浪费。

这两个例子，反映出对需求变化预估不足所带来的严重影响。从规划的角度讲，既要防止资源浪费，还要避免管理成本的提升，适度原则在这里要充分地体现出来。由于业务种类繁多，需求的变化与差异性加剧了规划的难度。我们建议以业务的体量为准，根据不同规模，采取 1C、1/2 C 或 1/4 C 作为地址段的分割单位，尽可能地将体量相同的业务放在一起。

例如，我们有五个业务模型，总共 190 台主机，如表 4-9 所示。

表 4-9 业务地址规划示例

| 业务类型 | 现有主机数 | 地址池           | 可用地址数 | VLAN |
|------|-------|---------------|-------|------|
| 业务 A | 30    | 10.0.0.0/26   | 62    | 101  |
| 业务 B | 50    | 10.0.1.0/25   | 126   | 102  |
| 业务 C | 70    | 10.0.1.128/25 | 126   | 103  |
| 业务 D | 10    | 10.0.0.64/26  | 62    | 104  |
| 业务 E | 30    | 10.0.0.96/26  | 62    | 105  |

从这张表中我们可以看出，业务 A、D、E 的体量是接近的，我们可以将这三者划分成一组，而把业务 B 和 C 划分成另一组。1/4 C 的切分已经足够小了，不要认为业务 D 现在只有 10 台主机，分配 62 个地址空间会造成浪费。一定要考虑到将来它也有扩容的可能。有些业务之间是有关联的，假定业务 D 扩容到 30 台主机时，业务 C 会增长到 200 台，那么业务 B 和 C 可采取 1C 的分配方式。当然，这完全取决于业务增长的可能性有多大。在规划子网时，力度一定要把握好，太小会增加管理成本，影响业务扩容。同时还要注意，不能让地址空间超出一个 PoD 的容量范围。

那么，有没有什么一劳永逸的解决方案呢？如果可以，我想就是使用大二层的技术，它可以让一个 PoD 容纳足够多的地址。在第二个示例的情况下，即便留有大量的空闲地址也不必担心。

### 4.3.3 VLAN 的规划

众所周知，在一个二层网络内，通信基本靠“吼”，设备寻址要借助 ARP 广播来实现。不过，广播域规模过大是有害的，它会引发巨额开销。但是二层网络又无法隔离广播域，而一旦发生广播风暴或 ARP 攻击，整个二层网络内的通信都会受到影响。三层设备可以隔离广播域，但是它的成本太高。因此，使用 VLAN（虚拟局域网）是一种普遍的解决方案。一个 VLAN 就是一个广播域。默认情况下，不同 VLAN 之间是相互隔离的。在 VLAN 内部，不论设备的真实位置是在哪里，它们彼此通信时，就好像处于同一个子网之中。而不同的 VLAN 之间，即便是同一个交换机上的两个端口，它们相互之间也是隔绝的。相比传



统的局域网技术，VLAN 不但能有效地抑制广播规模，提升网络的安全性，同时在管理上更加灵活，有利于减少管理开销。

### 1. 为什么需要 VLAN

由于 VLAN 的使用成本很低，用它可以很方便地在多个业务之间创建逻辑隔离。小卖部的柜台就是一个典型的业务隔离。交易支付发生在柜台前，而款项结算与货品存放则位于柜台的后面。从这个例子我们能够看到，和购买行为有关的所有内容并不是混杂在一起的。即使用最粗放的方式，我们至少也要将业务划分成前端和后端两大类。针对不同类型的业务，用 VLAN 将它们区分开来。

电商支付的方式远比小卖部更为繁杂。当用户在页面点击支付按钮时，中间要通过交易、支付、渠道网关等众多模块，最后才能到达银行系统。除了这条主线以外，可能还会有一些支线行为。例如，交易系统会查询用户资料。如果用户未实名却购买了大宗商品，则要进行风险识别（防止洗钱）。事实上，完成一项交易需要很多子系统之间的对接，每个环节要访问的数据库也都不同。因此，为了确保支付的安全性，必须把不同类型的业务规划到不同的子网当中去，以子网为对象来定义防火墙的安全策略。

数据库和应用不可能只有一类，它们之间是多对多的关系。因此安全策略千万不要采取点到点的模式。在设备数量较少的情况下，这样做没有什么太大的问题。但当设备数量超过一定规模时（比如 1000 台），工作量将非常惊人，而且大量的策略条目会让整体逻辑变得异常复杂。尤其是在业务紧急扩容的时期，网络工程师会发现，每天都有做不完的防火墙工单，其他事情什么也干不了。

而使用 VLAN，相当于启用了组间的策略，简化了多对多的关系，大大降低了管理的复杂度与成本。有一点要在这里说明，启用 VLAN 主要是针对应用区的设备，因为应用区是设备数量最多且种类最复杂的区域。

### 2. 如何规划 VLAN

举个例子来说明。假设我们有三个业务，业务 A、B、C 三者之间的规模比例是 1:1:2，我们可以按照表 4-10 的方式来进行分配。

我们前面也谈到过，机柜和设备的数量会受到 PoD 的限制。通常情况下，一个 PoD 最大能容纳的物理节点总数很难超过 400 个。所以，划分一个大的子网只会造成资源浪费。

表 4-10 业务 VLAN 规划

| 业务类型 | VLAN | 地址池           |
|------|------|---------------|
| A    | 101  | 10.0.1.0/25   |
| B    | 102  | 10.0.1.128/25 |
| C    | 103  | 10.0.2.0/24   |

同样，VLAN 的个数也不宜设置得太多。虽然我们强调使用 VLAN 去隔离不同种类的业务。但是，我们不可能给每一项业务都单独划分一个 VLAN，这样做不能给你带来任何好处，反而会极大地提升管理成本。我们应当按照宏观的业务类别来划分 VLAN，而不是用一个 VLAN 去对应一个业务。

比如，交易和结算显然是两种不同类型的业务，它们应当被分配到不同的 VLAN 之中。但是，交易里面可能还包含了很多不同种类的子项。这种情况，采用地址集进行切割可能是更佳适宜的形式。假设业务 A 下面包含了五个子项，每个子项的节点数较为平均。那么，我们可以按照表 4-11 中所示的样例来规划。

表 4-11 子业务地址集规划

| 业务类型 | VLAN | 地址池         | 业务子类 | 地址集            |
|------|------|-------------|------|----------------|
| A    | 101  | 10.0.1.0/25 | A.a  | 10.0.1.1-20    |
|      |      |             | A.b  | 10.0.1.21-40   |
|      |      |             | A.c  | 10.0.1.41-60   |
|      |      |             | A.d  | 10.0.1.61-80   |
|      |      |             | A.e  | 10.0.1.81-100  |
|      |      |             | 预留   | 10.0.1.101-126 |

VLAN 规划完成后，组间的安全策略一旦设定好，后续基本上是不用动的。如果今后扩容涉及新增地址，则需要将新增对象添加到现有的 VLAN 中。到时候虽然一个 VLAN 里会存在多个网关，但这并不影响我们的正常使用。

4.4 网卡绑定

Adapter Bonding 是一项在操作系统层面上实现的网络冗余技术。设置网卡绑定后，Kernel 会将多条物理链路在逻辑上组合成一条虚拟链路，为操作系统实现了网络层上的高可用。

4.4.1 网卡绑定模式的选择

大家如果研读过《RHEL Deployment Guide》这篇文章就可以得知，网卡绑定模式一共分为七种。接下来，我们详细地介绍这七种模式的特点。

1. Bond 0——Balance-RR 模式

Balance-RR 被称为轮询均衡。从组内的第一个 Slave 接口开始，以轮询方式将数据包按顺序依次发放。这和分发扑克牌的方式是一样的。实现该模式需要交换机配置端口聚合。它有一个非常大的问题，就是存在着数据包的传输乱序现象。尽管理论上，每个 Slave 的接

口速率和线路质量是相同的，但这并不能保证接收端就一定按照发送的次序来接收数据包。而乱序的数据包是不能用的，要么重发，要么让应用自己处理乱序的问题。但不管怎样，由此都会带来不必要的额外开销。

## 2. Bond 1——Active-Backup 模式

Active-Backup 无须交换机的支持，通常也称之为 Failover。它采用主备方式实现冗余的功能，就像两个人倒班一样，永远只有一个 Slave 接口处于工作状态。

## 3. Bond 2——Balance-XOR 模式

Balance-XOR 和 Balance-RR 有所不同。虽然同是均衡，但它对 Slave 接口的选择是有一定算法的。首先，它对源 MAC 地址和目的 MAC 地址进行异或，然后通过求模运算得到相应的 Slave 接口。其计算公式如下所示。

$$(\text{<source\_MAC\_address> XOR <destination\_MAC>}) \text{ MODULO } \text{<slave\_count>}$$

## 4. Bond 3——Broadcast 模式

Broadcast 属于多链路的镜像冗余，类似于 RAID 1，每一个 Slave 接口上跑的数据都是相同的。它的冗余能力最强，应用层根本感知不到线路故障。但这种模式和 Active-Backup 一样，严重浪费带宽。

## 5. Bond 4——802.3ad 模式

802.3ad 通过创建聚合组，统一了多个 Slave 接口的速率和工作模式。而且它的兼容性很好，是很多互联网公司的首选模式。Slave 接口的选举算法共有三种，你可以通过调整参数 `xmit_hash_policy` 来设定。

第一种算法是 Layer2，它对源 MAC 地址和目的 MAC 地址进行异或，然后根据 Slave 接口的数量完成求模计算，其计算公式如下所示。

$$(\text{<source\_MAC\_address> XOR <destination\_MAC>}) \text{ MODULO } \text{<slave\_count>}$$

第二种算法是 Layer3+4，它和 Layer2 的思路是相同的。只不过它的参考对象有所改变，不再是 MAC 地址，而是变成了 IP 地址和端口号，其计算公式如下所示。

$$((\text{<source\_port> XOR <dest\_port>}) \text{ XOR } ((\text{<source\_IP> XOR <dest\_IP>}) \text{ AND } 0\text{xffff})) \text{ MODULO } \text{<slave\_count>}$$

第三种算法是 Layer2+3，即基于 MAC 地址和 IP 地址进行计算，其计算公式如下所示。

$$(((\text{<source\_IP> XOR <dest\_IP>}) \text{ AND } 0\text{xffff}) \text{ XOR } (\text{<source\_MAC> XOR <destination\_MAC>})) \text{ MODULO } \text{<slave\_count>}$$

如果 `xmit_hash_policy` 未指定，默认算法是 Layer2。但 Layer2 仅仅基于二层作为定义域，对于带宽的利用是不够充分的。RedHat 官方手册中特别提到，Layer3+4 和 802.3ad 并不完全兼容，所以最好的做法是采用 Layer2+3。



## 6. Bond 5——Balance-tlb 模式

Balance-tlb 被称之为传输负载均衡，它不需要交换机的支持。出口流量会根据当前的负载情况动态地分配到每个 Slave 接口上，而入口流量仅由当前的 Slave 接口来承载。如果该接口发生故障，其他接口会顶替其 MAC 地址。

Balance-tlb 默认的分配原则是根据负载情况进行动态调整，如果你想禁用这个功能，可以通过关闭参数 `tlb_dynamic_lb` 来达成。此时，内核只会根据 Hash 的计算结果进行分配，而不是流量的负载情况。`tlb_dynamic_lb` 共有 1 和 0 两个值，1 代表启用动态分配，0 代表禁用。

## 7. Bond 6——Balance-alb 模式

实际上，Balance-alb 是 Balance-tlb 的升级版，它通过 ARP 协商，实现了针对 IPv4 的接收负载均衡的能力，也被称之为自适应负载均衡。Balance-alb 的底层驱动要支持硬件地址的改写。Bond 设备中的每个 Slave 接口都有一个唯一的硬件地址。ARP 协商后会选举一个 Slave 接口，它被称作 `curr_active_slave`，内核会使用它的地址作为 Bond 设备的硬件地址。如果该接口故障，会重新选举一个新的 `curr_active_slave`，Bond 设备的硬件地址也会被改写成最新的。

当本机收到 APR 请求时，底层驱动会拦截本地发出的 ARP Reply，将源 MAC 地址替换成 `curr_active_slave` 的硬件地址。针对不同的 ARP 请求，内核会使用不同的硬件地址，从而达到负载均衡的目的。

当本机发送 ARP 请求时，底层驱动会复制 ARP 报文并保存对端的 IP。当收到对端返回的 ARP Reply 后，它会抽取应硬件地址，并将一个已经分配好的硬件地址给对端返回。

使用 ARP 协商进行负载均衡带来的一个问题就是——每次使用 Bond 设备的硬件地址广播 ARP 报文的时候，所有对端在学到这个地址之后，都会把流量全部指向当前的 Slave 接口。

为了处理这个问题，底层驱动会给每个对端单独分配特定的硬件地址，然后给它们发送 ARP Reply 更新，以便达到重新分布流量的目的。此外，每当 Slave 的接口状态有变化的时候，都会导致接收流量的重新分配。而接收流量的负载是轮询分配在 Bond 设备中速率最高的那组 Slave 上。

另外，使用 Balance-alb 时要注意，参数 `updelay` 的值不能小于交换机的转发延时，这样才能确保发往对端的 ARP Reply 不会被交换机拦截。

关于绑定模式我们就介绍到这里，详细内容的描述请大家参见如下链接。

<https://www.kernel.org/doc/Documentation/networking/bonding.txt>

从带宽利用率的角度来看，802.3ad 的使用更为普遍，这也是我所推荐的绑定模式。接下来，我们以此为例，来说一说网卡绑定的具体实现。

## 4.4.2 网卡绑定的实现

实现网卡绑定的过程总共需要三步：创建虚拟网卡，配置物理网卡指向虚拟网卡，重启网络服务。假设我们有两块名为 `em1` 和 `em2` 的物理网卡，需要绑定成 802.3ad 的模式，虚拟网卡命名为 `bond0`。关于详细的配置方法，请读者参考下面配置文件中所示的内容。

```
[root@station101 ~]# cat /etc/sysconfig/network-scripts/ifcfg-bond0
DEVICE=bond0
ONBOOT=yes
BONDING_OPTS="miimon=100 mode=4 xmit_hash_policy=layer2+3"
TYPE=bond
BOOTPROTO=none
IPADDR=10.0.0.101
NETMASK=255.255.255.0
```

```
[root@station101 ~]# cat /etc/sysconfig/network-scripts/ifcfg-em1
DEVICE=em1
ONBOOT=yes
SLAVE=yes
MASTER=bond0
TYPE=Ethernet
BOOTPROTO=none
```

```
[root@station101 ~]# cat /etc/sysconfig/network-scripts/ifcfg-em2
DEVICE=em2
ONBOOT=yes
SLAVE=yes
MASTER=bond0
TYPE=Ethernet
BOOTPROTO=none
```

完成配置并重启服务后，我们如何验证 Bonding 是否生效呢？

老前辈们习惯通过插拔网线的方式来验证，但是如果你有很多个节点，或者人根本不在现场，这种方式就不太现实了。事实上，我们完全可以用更加简单的方式来验证配置的有效性。

首先，使用 `ip` 命令检查逻辑网卡和物理网卡的 MAC 地址，三个 MAC 地址应当是一样的，就像下面所示的这样。

```
[root@station101 ~]# for i in em1 em2 bond0; do ip link show $i ;done
2: em1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc mq master bond0
state UP qlen 1000
link/ether ec:f4:bb:c3:fa:44 brd ff:ff:ff:ff:ff:ff
3: em2: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc mq master bond0
state UP qlen 1000
link/ether ec:f4:bb:c3:fa:44 brd ff:ff:ff:ff:ff:ff
6: bond0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
link/ether ec:f4:bb:c3:fa:44 brd ff:ff:ff:ff:ff:ff
```

然后，检查状态文件 `/proc/net/bonding/bond0`。看 MII Status 是否是 up 的，并观察 Aggregator ID，Slave 接口的 ID 和聚合组的 ID 应保持一致，而且聚合组内的 Number of ports 的数值应等于 Slave 接口的数量。下面的输出中，由于我们绑定了两个端口，所以 Number of ports 的数值为 2。

```
[root@station101 ~]# egrep "Agg|Mode|Policy|ports|Status" /proc/net/bonding/bond0
Bonding Mode: IEEE 802.3ad Dynamic link aggregation
Transmit Hash Policy: layer2+3 (2)
MII Status: up
Aggregator selection policy (ad_select): stable
Active Aggregator Info:
    Aggregator ID: 1
    Number of ports: 2
MII Status: up
Aggregator ID: 1
MII Status: up
Aggregator ID: 1
```

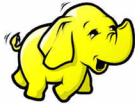
如果协商失败最主要的原因是连线错误，这种物理层的错误确实很让人郁闷。关于这一点，我体会很深。我敢说至少有一半的可能性。另外，还有一种可能需要大家注意。RHEL 7 的语法检查比 RHEL 6 要严格。在 RHEL 6 上，bondX 的类型可以写成 TYPE=Ethernet，但是在 RHEL 7 上这样做会出错，只有 TYPE=Bond 才能生效，否则协商失败。这是我使用老版本 Cobbler 安装 RHEL7 时获得的一个经验。

## 4.5 本章小结

本章我们讨论了网络规划细节上的种种问题，并就网络资源规划、网卡绑定等内容进行了详细的分析讨论。从资源规划的角度讲，既要防止资源浪费，同时还应注意降低管理成本的开销，体现出适度、实用的原则。网卡绑定推荐读者使用 802.3ad layer2+3 的模式，这是目前主流互联网比较常用的解决方案。

说完了网络的那点儿事，我们还是要回归到系统运维实践中来。下一章，我们将一同探讨服务器硬件选型的话题。





## 第 5 章

# 服务器硬件选型

硬件平台是支撑生产系统运行的基础设施。随着企业的不断发展，应用负载和数据量也在日益增加，只有搭建一套性能优良、稳定可靠的硬件平台，才能保障生产系统高效、稳定和安全地运行。

服务器是硬件平台中最为关键的部分，它既是硬件平台中数量最多的设备，也是直接承载业务及数据交换的关键环节。本章将阐述与服务器相关的硬件选型工作。

### 注意：

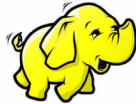
由于硬件产品有着更新换代快、生命周期短的特点，而本书在出版时，书中内容极有可能存在一定的滞后性。因此，本章重点旨在介绍硬件选型过程中的思路，但所涉及的硬件产品不作为最终的推荐方案。

## 5.1 如何选择合适的硬件配置

硬件平台系统的建设与业务发展的规模之间有着紧密的联系。下面，我们就来看一下，硬件平台系统的建设都有哪些特点。

在硬件平台系统的建设初期，服务器的数量和维护人员都很少，应用部署相对比较集中。假使某一个节点的服务器发生硬件故障，对业务会产生比较大的影响。而频繁发生硬件故障也会使维护人员的工作压力增大。这个时期，我们对于硬件设备的可靠性要求是最高的。所以，应当选择功能强、故障率低、日常维护简单便捷的产品。建议选择两家供应商，既减少产品多样化带来的麻烦，也防止被单一产品绑架。

而平台建设的中期阶段，是业务扩张速度最快的时期。此时服务器数量明显增多，成本压力很大。企业在发展到一定规模后，为了成本控制与合规性，会引入更多的供应商参与入围竞标。这时就会逐步形成异构平台的乱局，运维团队将面临管理成本的上升。那么，这就需要我们采取应对手段，做好充分准备，去打这场异构融合的无硝烟战争。



当平台建设进入后期时，服务器的数量已经非常庞大了，应用也应当完成了分布式部署的建设工作。单节点故障对于业务的影响已经被弱化甚至消除了。接下来，我们需要考虑的问题有两个：如何精简服务器的硬件配置来降低整体成本，以及如何处理生命周期完结后的老旧设备。

### 5.1.1 选型的总体原则

在硬件平台建设的过程中，服务器的选型设计应遵循如下基本原则。

- ❑ 统一规划，明确应用系统在规划期内的规模，对整个应用系统的模块、用户、流程进行分析；确定总体需求，从而定义硬件平台的架构和配置。
- ❑ 高可靠性，要求硬件平台长期能够稳定地正常运行。服务器的适用性强，故障率低。在产品生命周期内，确保供应链可靠和备件货源充足。
- ❑ 操作便利性，日常操作简单方便，可维护性强，满足大批量集中管理的需求；对于设备运行状态提供有效监控，在出现故障事件时能够及时预警；分析日志的内容输出详细，可以留存、转发和导出。
- ❑ 弱化纵向扩容能力，要在硬件产品的整个生命周期内，充分考虑业务需求的延展，硬件配置应当在整个生命周期内都满足业务需要；生命周期结束后，设备升级方案应当采取整机替换，而不是部件升级。
- ❑ 符合未来发展走向，要根据 x86 未来应用的发展特点来选择合适的架构，主要宗旨是弱化单节点影响，强调架构简单和分布式的部署方式。
- ❑ 产品线平滑过渡，选择主流的产品，确保更新换代时可以平滑过渡，防止出现产品断层，硬件平台选型前后差异化过大，会带来不可估量的风险。
- ❑ 高性价比，质优价廉。

### 5.1.2 选型中值得注意的地方

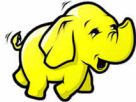
#### 1. 内存

内存没有什么太多好讲的，知名品牌也就是三星（Samsung）、现代（Hyundai）、金士顿（Kingston）这几家而已。内存选用的重点是要确保其参数与你选择的 CPU 是相匹配的。Intel 官方在针对每一款产品的描述中，都会提供内存匹配的指标要求，如图 5-1 所示。

Memory Specifications

|  |         |                            |                     |
|--|---------|----------------------------|---------------------|
| Max Memory Size (dependent on memory type) ? | 1.54 TB | Memory Types ?             | DDR4 1600/1866/2133 |
| Max # of Memory Channels ?                   | 4       | Max Memory Bandwidth ?     | 68.3 GB/s           |
| Physical Address Extensions ?                | 46-bit  | ECC Memory Supported ? ? ? | Yes                 |

图 5-1 对于内存匹配的指标要求



## 2. 网卡

低延时、资源占用少、稳定高效是我们选用业务网卡的主要要求。可以根据实际需求选择千兆或者万兆的网卡。比较有代表性的就是 Intel 的 i350 和 x520。如果选择万兆网卡，不一定非得采购光模块，也可以使用 AOC 线缆来替代。网卡最好是 On-Board 的，这样就可以通过 BIOS 去控制网卡的启用或禁用，而外接式的网卡是否能够被 BIOS 所支持，就要看厂商能不能提供支持了。另外，对于带外管理卡必须要有独立的网口，不要使用 NCSI 去替代。

## 3. 服务器规格

尽可能缩减服务器的规格。如果能使用 1U 的服务器，就坚决不用 2U 的。2U 规格的空间是为了加载更多配件而设计的，如果采购 2U 的理由只是考虑设备升级扩容，就完全没有必要了。生命周期内，硬件配置应当符合预期要求才对。服务器质保期只有三年，三年以后的硬件性能会比现在提升很多，续保也需要不少的费用，与其升级配件不如直接换新机器划算。服务器越小越轻，维护起来就越方便。而且数据中心只按机柜计费，小规格的服务器可以上架更多的设备。提升机柜的设备上架率，可以有效降低总体成本。

## 4. 磁盘的容量与数量

如果同是 RAID 的情况下，300GB×8 和 600GB×4 的磁盘配置应该如何选择呢？前者的读写性能好，而后者的故障率和成本更低。对于一般应用来说，如果对于 Throughput 和 IOPS 没有那么大的需求，后者的选择更好一些。性能并不是越高越好，没有使用到的资源就是浪费。在硬件选型的过程当中，应当始终遵循配置最简化的原则。部件越少，出问题的概率就越低，同时成本与能耗也会相应降低。

## 5. 低内存大硬盘的需求

类似 4GB 内存加 300GB 存储的这种需求真的让人很尴尬。假如这种需求量还特别大，则会带来很多麻烦。因为采购这样配置的设备成本并不低，而且会挤占数据中心的存放空间。使用虚拟机的话，这种不对称的配置又会造成大量的资源浪费。

由于这一类应用基本上都属于低负载、慢响应的服务，建议在存储上使用 NFS 的方案来为其提供空间，存储可以用服务器来替代，使用 iSCSI 方式连接。

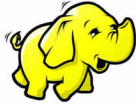
## 6. 备件供货保障期

最早，我们采购合同里对备件的供货周期是七年，其实没有太大必要。应当说，只要备件供货的保障期和设备质保期相等就可以了。设备在质保期即将结束之前，就应当完成迁移工作，因此没有必要保留更多的备件。另外，在质保期内，厂商都可以确保备件供货，没有必要自己去购买备件，增加资产管理的成本。如果有一些机器允许超期服役（例如测试设备），备件可以从其他退役设备上拆件做替换，反正是过保的，没有必要买新的。

## 7. 过保服务器的处理

因为服务器的质保期为三年，大部分应用应当在二年半左右的时间做好迁移的准备工





作。服务器在这个时候如果可以完成下线，有两个比较好的处理途径。第一是回收，因为设备还没有过保，此时资产的残值相对比较高，变卖后的资金可以用于新一轮的采购。但是注意要做好数据脱敏工作，由有资质的厂家来完成相应的回收。第二是废物利用，可以将退役的设备留给测试或者内部办公使用，也可以拆件成立备件资源库。

## 5.2 怎样的一款服务器产品才算是优秀的

对于一款服务器产品而言，我们可以从三个角度去评估它——质量、功能和服务。

质量是产品可用的第一要素，这里主要是指硬件的故障率，这个数值应当低于 2%，一些厂商的硬件质量甚至可以做到低于 1%。不要小看这个数字的变化，对于海量模式的硬件平台，基数越大，差距效果就越明显。假设服务器总量是三万台，多出 1% 就意味着平均每天会多触发一次故障事件。即便是更换硬盘的维修，也会对生产系统的运行产生一定影响。故障率如果控制不住，那么 SLA 的承诺就是一纸空谈。

质量也直指另外一个重要指标，那就是性能。在这里，我们把产品性能也纳入质量里面去。为什么这样讲呢？如果一个产品的性能很差，意味着它的可用性就很差，一个不可用的产品，从概念上讲基本就等同于质量不合格。一个性能极差的产品基本上就是不可用的，对生产业务的正常运行是非常不利的。

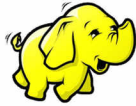
功能是另外一个很重要的影响因素。如果说质量关系到业务的正常运行，那么功能就关系到业务的高效维护。一般所谓的功能主要是指服务器的带外管理功能。因为在硬件配置方面，除了 RAID 卡和电源以外，能够相互一较高下的地方并不太多。但是服务器的带外管理功能确实可以有效地拉大不同产品之间的距离。

质量和功能在我们的技术评估中是占有很大比重的，而服务部分的比重会相对偏轻一些。服务好不如质量好，手册好不如产品好。如果一款产品，质量可靠有保障，使用简单不求人，那么谁还会需要售后服务和说明书呢？如果产品的质量跟不上去，功能又有缺陷，那么服务再好也是没有意义的。相反的，如果产品功能强大且质量过关，我们反而很少会使用到售后服务。

在海量模式下的运维场景中，甲方都有自己专门的运维团队。当触发任何紧急事件时，第一时间都需要运维团队自己解决。我们不可能像过去传统的系统集成那样，把所有工作交付给厂商来完成。这涉及一个时间成本的问题。一个线上系统发生故障了，难道你要我打 800 开 Case，再等着厂商派工程师出现场吗？这显然是不可能的。业务就算停止一分钟，损失都难以估计，你根本就等不起。只有像硬件维修或是技术咨询这类不紧急的问题，我们才会依靠厂商来支持。

另外，服务是一个长期积累的过程。一个厂商的服务好与坏，在短期内是很难做出评判的。对于那些以前根本就没有使用过的产品，服务这一项也仅能通过测试阶段的售前表





现来看。这也是不能把服务占比过重的一个客观因素。

### 5.2.1 带外管理有多重要

做系统运维的同学会经常提到带内管理与带外管理这两个名词。所谓的“内”“外”之分，就是指管理通信链路和业务生产通信链路之间的关系。如果我们使用业务所在的链路进行消息传递和管理，我们就称之为带内（In Band），反之就是带外（Out of Band）。

我们日常维护生产环境，主要是通过带内网络进行管理。所依靠的手段无非就是 RDP、VNC、SSH、TELNET。但是，这些服务都运行在操作系统上面，并且通过网络远程访问，其中存在很多不稳定的因素。比如硬件故障，操作系统崩溃，或者是人为操作失误导致系统无法访问，等等。由此看来，带内管理这条通道是很脆弱的。我们需要使用另外一个备用的手段，来确保我们对设备和操作系统的控制权。

带外管理是完全独立于现有生产环境的，从硬件接口、网络链路，再到存储和操作系统都是单独分离出来的。带外管理系统存放在一个很小的控制芯片上面，里面是一个经过修剪的、只读的最小系统环境，通过单独的接口与网络去访问。所以它的可靠性比带内管理网络要高得多。只要控制芯片加电且带外的网络正常，我们就可以始终把控制权牢牢地掌握在手中。

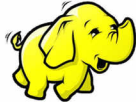
当带内管理网络崩溃时，我们依旧可以凭借带外管理提供的虚拟控制台，远程登录生产系统的本地 console 界面，这相当于在机房里面直接接上 KVM（Keyboard-Video-Mouse）设备，是我们处理故障最有效的保障手段。除了断后之外，带外管理也扮演着开路先锋的角色。在设备刚刚上架加电的时候，我们是没有带内环境的，系统的安装部署、服务器的开关启停都离不开带外管理。

也许在一些人看来，带外管理不过是提供了一个远程的虚拟控制台而已。但实际上，它所能完成的任务远远不止这些。优秀的带外管理可以说是提供了所有你在本地操作面前能做的一切功能，甚至还有额外的增值项目。我们可以借此获取详尽的硬件清单配置列表，收集监控数据信息，设置 BIOS 参数，甚至操控硬件。

### 5.2.2 异构平台融合能力

从管理角度讲，只使用一家厂商的产品，对于资产的统一管理与配置是有利的。如果出货量大，双方相互之间还可以签署框架协议，进一步推动价格成本控制和产品定制化。这对于平台初期的快速建设是有一定帮助的。不过，当平台规模从溪流模式发展到江河模式或者海量模式的时候，一些政策法规不允许只用一家的这种采购形式，同时单一化产品也存在品牌绑架的风险。这个时期，就会突然涌现出许多不同品牌，它们都有可能在未来同时入驻到我们的服务体系当中去。由于来自不同产品之间的差异会带来多样化管理的难题，这就对服务器的异构平台融合能力提出了严苛的要求。我们不希望看到因为产品差异





化而增加运维的管理成本，因此，必须弱化这种差异效应，让运维团队的成员感受不到不同产品之间的切换与变化。

支持并使用标准的公有开放式协议是异构融合的关键。私有协议不管做得多好，对于一统天下都是没有任何帮助的。除非你没有竞争对手，或者你的私有协议能成为公认的标准。

IPMI 协议尽管发展使用了将近 20 年的时间，可以方便地为用户提供电源控制、传感器监控等通用型功能，但是它已经是一个落后于时代的产物了。作为 x86 平台的工程师，我们一直都羡慕小型机上面有专门获取硬件信息的命令。而 IPMI 对于这方面需求的发展一直难有作为。事实上，一些厂商像 Dell、联想在 IPMI 上也有 oem 接口，但是 IPMI 所做的工作实在是太有限了，我们需要一个新的方案来解决异构平台上的管理难题。

WS-Management，全称叫作 Web Services-Management，是 DMTF 组织基于 SOAP (Simple Object Access Protocol，简单对象访问协议) 制定的一种开源标准。该标准致力于在不同的 x86 设备厂商当中，提供一种 IT 基础架构信息访问与修改的统一接口。这对于那些支持该标准的厂商来说，会给用户有效地管理资产配置工作提供极大的帮助。例如，我们有很多来自不同厂商的服务器设备，如果它们都能够很好地支持 WS-Management 标准，那么就可以通过 wsmancli 工具，统一采集或修改所有服务器的硬件配置信息，而不必因为私有化工具分治的问题而形成多头管理的局面。AMD、Dell、Intel、Microsoft 等知名厂商都是该项标准组的成员。

这里我们就目前已经实际应用了 WS-Management 协议的 DELL 服务器为例。如果需要获取网卡的 MAC 地址，可以使用如下命令实现。

```
# wsman enumerate \
http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/root/dcim/DCIM_NICView \
-h <HOST> -V -v -c dummy.cert -P 443 -u <USER> -p <PASSWORD> -j utf-8 -y basic \
| grep PermanentMACAddress | sort

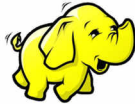
<n1:PermanentMACAddress>EC:F4:BB:C3:E9:94</n1:PermanentMACAddress>
<n1:PermanentMACAddress>EC:F4:BB:C3:E9:95</n1:PermanentMACAddress>
<n1:PermanentMACAddress>EC:F4:BB:C3:E9:96</n1:PermanentMACAddress>
<n1:PermanentMACAddress>EC:F4:BB:C3:E9:97</n1:PermanentMACAddress>
```

使用 WS-Management，必须先在客户端上面安装名为 wsmancli 的软件包。DELL 在这方面走得还是非常靠前的，官网上不但给出了 wsman 的使用实例，同时还在 git 上面提供了一整套范例脚本。具体内容请读者自行参考如下链接。

- 1) WS-Management: <http://www.dmtf.org/standards/wsman>。
- 2) wsman 的范例参考: <http://en.community.dell.com/techcenter/systems-management/wiki/4374.how-to-build-and-execute-wsman-method-commands>。
- 3) 范例脚本下载地址和说明:







- ❑ <https://github.com/dell/recite>;
- ❑ <http://en.community.dell.com/techcenter/systems-management/w/wiki/3757.recite-interactive-ws-man-scripting-environment>;
- ❑ [http://en.community.dell.com/techcenter/extras/m/white\\_papers/20066176](http://en.community.dell.com/techcenter/extras/m/white_papers/20066176);
- ❑ [http://en.community.dell.com/techcenter/extras/m/white\\_papers/20066181](http://en.community.dell.com/techcenter/extras/m/white_papers/20066181)。

除了 WS-Management 以外,类似的标准还有 Redfish,最新版本是 1.2。它是一个通过 RESTful 接口利用 JSON 数据来实现的集成解决方案。Redfish 是一个更加轻量级的协议。比起 WS-Management 来,它同样借助了 HTTP 模式,但传输数据更少,协议层更加简单,所能够支持的成员也更多。国内一些厂家已经在推动 Redfish 项目的进程了。具体的详细内容可以参考如下链接: <http://www.dmtf.org/standards/wsman>。

另外一点就是兼容性。兼容性的优势将使产品在未来异构融合的竞争处于有利的位置。我们可以回顾一下历史,看一看 WinZip 为什么会输给 WinRAR。当年 WinZip 是压缩软件界的老大哥,而 WinRAR 只是初出茅庐的毛头小子而已。不过,WinRAR 在推广自己压缩率更高的 rar 格式的同时,也兼容了 zip 格式。而 WinZip 却不愿意把 rar 格式纳入自己的帐下。这两种策略最终形成了两种完全不同的结果。尽管后来各式各样的压缩软件如雨后春笋般出现,但都无法撼动 WinRAR 霸主的地位。原因就在于 WinRAR 能够谨记 WinZip 失败的教训,不断地兼容后来者的各项功能,稳固了自己的江山。

我们就以虚拟控制台为例。绝大多数服务器的虚拟控制台依旧是通过 Java 程序来实现的。而 HP 提供的 C/S 模式的工作效率显然要比 Java 高很多,并为本地登录提供了冗余手段。按理说,HP 能做到这一步,在众多厂商里面已经算是很领先的了。但是 DELL 的思维模式却显得更加前卫,它把 VNC 服务直接嵌入带外管理模块中,在本地登录的时候使用 VNC Viewer 就可以了,而不是像 HP 那样要安装一个私有化的客户端。使用开放标准的 VNC 协议的优势就在于没有开发成本,而且它的通用性和可行性都很强,在未来异构平台融合的大背景下具有很大的竞争优势。在写这本书的时候,我已经在和华为、联想、浪潮的销售与技术团队的技术交流中,建议厂商尝试在带外管理中嵌入 VNC 服务。这也许将在未来形成一种趋势,我个人希望借助宣讲和推动,为业界产品的异构融合迈出坚实的第一步。

### 5.2.3 完善的信息数据展示

信息展示分为静态和动态两个部分,静态信息是指硬件清单的配置信息,动态信息是指部件状态的实时监测数据。

硬件清单的配置信息用于资产管理和安装部署之用,采购的机型不止一种,所以需要通过对硬件清单列表的详细内容对服务器进行辨识,同时也有助于我们检查供货设备硬件配置的准确性。硬件配置清单列表中应当尽可能地包括如表 5-1 所示的内容信息。

表 5-1 Business-IP 的对应关系表

| 选 项    | 细 节                             |
|--------|---------------------------------|
| CPU    | Vendor、Type、Frequency、Amount    |
| Memory | Vendor、Type、Size、Amount         |
| RAID   | Vendor、Type、RAID info           |
| PD     | Vendor、Type、Size、SN、Amount      |
| LD     | Pool info、Size                  |
| NIC    | Vendor、Type、Port、MAC、Amount     |
| Power  | Power、Amount                    |
| OOB    | MAC、Firmware                    |
| Server | Vendor、Type、SN、Service Tag、BIOS |

部件状态的实时监测数据分为两种。第一种是硬件的健康状态，表明它是好的还是坏的。第二种是温度、电压、电流、风扇转速、能耗等传感器的实时读数。

## 5.2.4 软硬件环境兼容性

软硬件环境的兼容性的优劣，决定了一款产品应对使用场景的灵活性和适用性。如果适用性太差，就很难去推广到大规模、多样化的应用场景里去。

### 1. 浏览器兼容性

基于 B/S 架构的图形化管理是带外管理最早的表现形式。说到 B/S 架构，我们就不得不提到浏览器这个话题。近年来，Chrome 和 Firefox 确实比较流行，但是我希望更多情况下，产品对于 IE 浏览器的兼容性还是要多考虑一些。毕竟 IE 是 Windows 原生的，而很多工作我们还是要基于 Windows 平台来实现的。

### 2. JRE 兼容性

绝大部分产品都是基于 Java 开发的，所以 JRE 是远程虚拟控制台和一些私有化管理工具必不可少的运行支持环境。然而不幸的是，因为厂家众多，产品的生命周期也不尽相同，一些产品只能运行在老版本的 JRE 上面，而另一些产品则必须使用最新版本的 JRE。一个平台下同时使用多套 JRE 环境是很麻烦的。一个兼容性差的产品就像一个不合群的人一样，会受到别人的排挤。我们对待产品也是一样，少数要服从多数，新来的要入乡随俗，只有那些能够向下兼容的产品才会被留下来。

### 3. 协议兼容性

比较常用的协议有 FTP、LDAP、HTTP、HTTPS、NTP、SMTP、SNMP、SSH、rsyslog 等，而且类似 SNMP、SSH 等还有多个不同版本。对于产品所支持的协议的种类与版本，肯定是越多越好。



#### 4. 硬件及驱动程序兼容性

这个问题主要体现在和部署系统的配合上面。例如，因为硬件特殊或是驱动程序的问题，使得部署系统无法识别，从而导致安装失败。这种问题严格来讲，不能全都赖在厂商这边。但是用户作为甲方，是有一定话语权的。能够解决问题是最好不过了，但如果解决不了，或者解决成本太高，有可能用户会因此令你出局。

### 5.2.5 用户体验

产品生产出来就是给人用的，评价一件产品的优劣往往是很主观的，这就是我们常说的用户体验。对于产品研发来说，良好的用户体验是至关重要的。如果你把产品当成一个人，当你和这个人接触过一段时间之后，他或多或少地都会给你留下一个印象，那么你对这个人的评价就来源于你对他的印象。当然，这个评价可能是好的，也可能是差的，就是所谓的用户体验。一个获得好评的人，双方以后相互往来的可能性就会增加，反之人们就会疏远他。同样的，如果一个产品的用户体验很差劲，那么用户最终的感受就是——“我再也不想见到你了”。

用户体验是在用户使用产品的过程中建立起来的感受。对于一个界定明确的用户群体来讲，其用户体验的共性是能够经由良好的设计来实现的。产品设计的中心原则就是以用户为中心、以人为本。

#### 1. 响应时间

鲁迅先生说过：“生命是以时间为单位的，浪费别人的时间等于谋财害命；浪费自己的时间，等于慢性自杀。”不错，我想这世界上没有比等待更令人烦躁的事情了。如果点击一个页面之后，等了半天没有反应，实在是很令人沮丧。在毫无提示的情况下，等待会让人不知所措。尤其是在我们急于得到结果的时候，这种慢吞吞的节奏令人感到非常不合拍。

#### 2. 用户界面

我们不得不承认这是一个看脸的时代，“颜值”在产品里面也显得十分重要。

我认为 DELL 的导航菜单做得最为出色。它的界面完全符合 Web 页面的浏览习惯，隐藏类细节信息都会以加号展开的方式展现。操作方式也十分便捷，只需要点击两次鼠标，就可以完成大多数的操作与信息展示。DELL 的所有操作和信息展示都是在同一个页面下完成的，而不会弹出一堆 pop 窗口，也不会利用超链接给你跳转到另一个页面去。Intel DELL 服务器的带外管理界面如图 5-2 所示。

H3C 也很有特点，从管理界面和菜单结构上，还是可以看到很多 HP 的影子的。不过相比 HP，我认为 H3C 还是有一些自己的特色在里面的。比如，Overview 这个总览页面的右下角，汇集了很多常用的功能，点击对应的快捷方式可以方便地跳转到相应的功能当中去。H3C 的管理界面如图 5-3 所示。





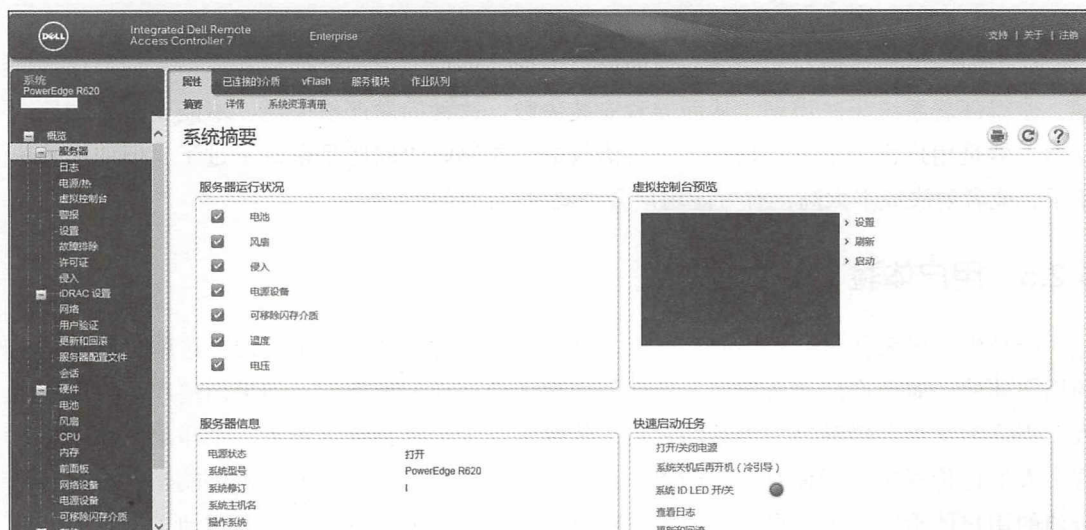


图 5-2 DELL 服务器的带外管理界面

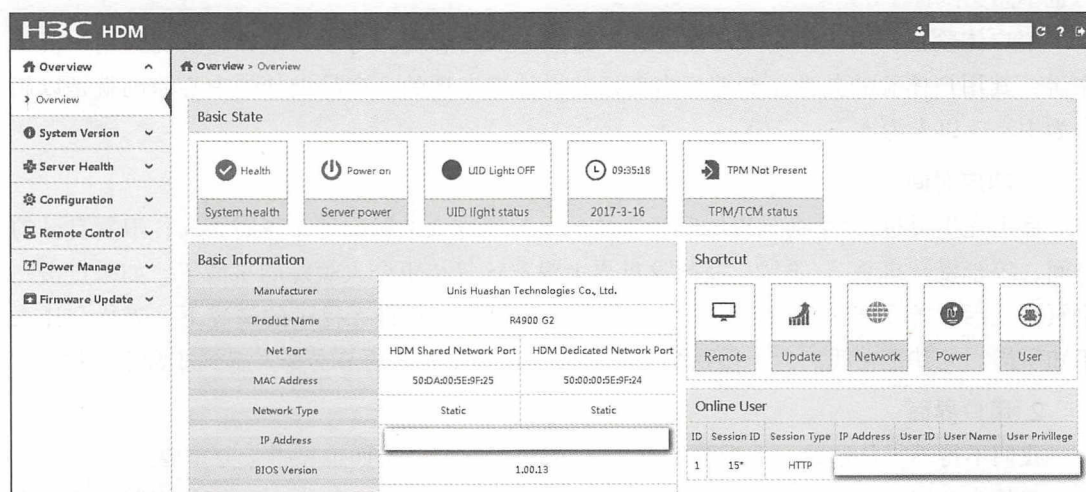


图 5-3 H3C 服务器的带外管理界面

本来对于超链接，我还是不太推荐的。因为很多产品在设计的时候，乱用超链接，看上去很方便，实则不停地跳转页面，把用户搞得晕头转向。而 H3C 的这个超链接设计却是可行的。它把所有常用功能都放到了一起，相当是一个门户页面的结构，还有类似 Dash Board 的关键数据展示。这个设计思路还是很用心的。

### 3. 产品逻辑

产品设计的优先级顺序应该是：逻辑性 > 稳定性 > 性能 > 功能。

产品的逻辑性非常重要，一个产品的使用逻辑有问题，往往就带来极其糟糕的用户体

验。使用一个逻辑混乱、结构模糊的产品，对用户就是一种折磨。

在这里，我举两个例子。

第一个例子是我们在调试一款产品的功能时发现的。我们要求服务器厂商提供的产品，在系统配置上必须有图形和命令行两套接口。图形主要适用于调试使用，而命令行则有利于我们日常的批量设置。我在配置一个新的参数时，发现在命令行里找不到这个参数，而且文档里面也没有提及。之所以我找不到，是因为图形界面里菜单的组织形式和命令行树形的组织形式不完全一样。这样一来，我没法根据图形界面的菜单去定位它在命令行树形结构中的位置。最后我不得不采取了一个笨办法，把根目录下的所有属性都列了出来，再通过过滤关键字的方式来搜索答案。

第二个例子是在一次测试过程中遇到的。我个人负责产品测试，另外一个同事负责安装部署测试。有一款产品，我先对它做了功能评估的测试，然后转给同事去测试安装部署。过了一会儿，同事转过头跟我说，这个产品不行，好多硬件信息根本看不到。我笑着告诉他，其实信息是有的，只是它们因为害羞躲起来了。那些硬件信息的细节全部都被隐藏起来了，而且隐藏得太好了，导致我的同事根本没有发现。因为它们的打开方式并不统一，窗口、标签和链接各式各样。如果不仔细看，就会给人造成信息缺失的假象。

关于产品的逻辑问题还有很多，但我想原因只有一个，归根到底是设计人员从来不用自己设计的产品造成的。

## 5.3 产品测试那些事儿

### 5.3.1 测试前的准备工作

测试环境的一致性，是测试工作中很重要的一点，无论再怎么强调都不过分。就像你购买一件商品，假如柜台上摆放的样品和你最终购买的商品不是同一件东西，那么这个样品展示就是一种欺诈。同样的，如果一个测试环境和最终生产的使用环境不一致，那么任何测试结果都是不可信的。

我经历过一件印象比较深刻的事情。我当年干系统集成时做了一个 VDI 的项目，销售给客户推荐了一款 Thin Client，这款产品价格比较便宜，尽管它在实验室里面运转得非常好，但是一到现场就出现各种各样的问题，客户对此抱怨很大。最后究其原因有两个：第一是送测产品本来就是旧的返修货；第二就是客户实际的使用环境要比实验室恶劣得多。

一致性这么重要，但是在实际测试过程中，我们仍然发现很多测试存在不一致的情况，尽管之前我反复强调要求，但是最后送测的设备还是存在这样或那样的偏差。为什么会出现这种问题呢？一方面是大部分厂商都是代理商送货，传达的时候漏掉了，这一点实在是太不应该了。另一方面是没有货，例如 SSD 磁盘的数量不足。

对于这种情况我建议，首先要在测试之前给供应商发送测试机型的配置要求，还要给

予供货商充足的送货周期，测试一定要提前至少 2 个月以上，不要临时抱佛脚。

设备到位，要对硬件配置再做一次复查。

❑ CPU 的配置：C State、C1E、虚拟化以及超线程的开启等配置；

❑ RAID 卡的配置：RAID Level、缓存、回写策略；

❑ 硬盘的配置：单盘容量、品牌型号、数量等；

❑ 内存的配置：单条容量、数量等；

❑ 网卡的配置：品牌型号、数量等。

不同的机型应当各送测一台，对于一套设备携带多种配置的方案，我个人是不建议的。这样做对厂商来说比较省事，但是对测试是不利的。因为一台设备反复更换硬件，意味着没有横向比较的条件了。如果可以，应尽量避免这种情况发生。

这里面所提及的所有硬件配置应当如数按照要求提供，如果一定要打折扣，比如 SSD，我会接受减少数量或调整容量。因为 SSD 本身能耗小，不会过多影响到能耗测试，而性能测试可以横向比较不同品牌相同配置的数据结果就可以了。

### 5.3.2 部署系统测试

部署系统对接的测试工作是首先需要完成的。所谓部署系统对接测试，是指我们的部署系统是否能够顺利地识别出测试设备的硬件，能够抽取所需的全部信息，并完成操作系统的安装。这项测试的失败，意味着今后无法完成批量系统交付的工作。我们不可能到后期，慢慢处理部署系统的各种细节问题。所以，这项测试应当最先完成。只有通过了这个环节的测试，后面的测试工作才有意义。

相对于系统的安装和硬件识别，硬件信息抽取是一个比较大的问题。硬件信息应当能够从带外管理里面抽取，而不是等到操作系统安装完成后，再到系统里面去找。即便是手工安装，在安装一个操作系统之前也是需要硬件配置有一个基本了解的。而且不同的硬件配置机型所对应的安装要求也可能不一样，所以在安装之前收集信息是非常重要的。然而，一些产品因为硬件信息设备清单列表做得不够到位，在带外管理本身就看不到各种信息。更不要说完整的硬件清单列表了。

### 5.3.3 产品功能性测试

对于一款服务器来说，产品功能性测试主要是指针对服务器带外管理功能的评测。在这方面不得不说的是，各家产品之间的差距还是非常大的。具体内容，大家可以参看 5.2 节关于带外管理能力的详细描述。

这一步评测也是至关重要的，一个产品带外管理能力的强弱，决定了我会不会大量采购这个产品。在海量运维的模式下，没有优秀的带外管理能力是根本不可想象的。毫不夸张地讲，一个人不具备这样的能力，就不能胜任海量运维的工作。一台服务器不具备这样



的能力,则不能满足海量运维的需求。长此以往,必将被市场淘汰。

### 5.3.4 能耗测试

在介绍能耗测试之前,我们必须弄清楚两个名词的概念,那就是 Turbo Boost 和 TDP。

#### (1) Turbo Boost

Turbo Boost 是英特尔的睿频加速技术。这项技术可以理解为自动超频。当开启睿频加速之后,处理器会自动加速到合适的频率,并根据当前资源的占用情况实时做出调整,以此来应对多任务情况下不同进程对于性能的要求。

当某个核心负载增大时,处理器会检测 TDP、 $T_{case}$ 、 $I_{cc}$  等读数后,根据资源利用率,降低闲置核心的电压与频率,转而帮助负载高的核心提升电源与频率。在进行动态调整的时候,必须确保 TDP、 $T_{case}$ 、 $I_{cc}$  等数值不会超过系统最初设计的高限。处理器内部的数字温度传感器(DTS)负责检测内核的温度,在需要时降低温度,对频率适时调整,让所有操作都在安全范围之内。

#### (2) TDP

TDP 全称是 Thermal Design Power,翻译过来就是散热设计能耗。它是为散热系统设计提供参考用的。Intel 对 TDP 的官方解释如下:

TDP: Thermal Design Power, A power dissipation target based on worst-case applications. Thermal solutions should be designed to dissipate the thermal design power. System and Maximum TDP is based on worst case scenarios. Actual TDP may be lower if not all I/Os for chipsets are used.

理论上处理器在满负荷工作状态下,它的温度应当是最高的,那么它所散发出来的热量也是最大的。此时的这个状态就是文中所讲的最糟糕、最坏的情况。这时候,散热设备必须有能力应对这种状况,确保散热效果,不至于让处理的温度超过最初的设计值。

如果 TDP 是 90W,意味着你的解决方案必须有能力驱散等同于 90W 用电能耗所带来的热量才行。否则当你的处理负载达到满载时,有可能出现散热不良导致处理器烧毁的情况。如果你的没有使用到很高的电力,就没有实际利用上 TDP 设计的全部。如同 750W 的电源不意味着设备一定需要 750W 的电力一样,这个 TDP 也是一个最大值。实际发热量应当控制在 TDP 最大值的范围内。

所以,TDP 是热能耗的安全阈值,绝不是处理器的用电能耗。请注意,这是两个完全不同的概念。

在探索资料的过程中,笔者在 Intel 官方网站的描述里也发现了一些令人迷惑不解的地方。在产品介绍中关于 TDP 名词解释,当你用鼠标点击问号后,里面的描述如图 5-4 所示。

按照这个解释,TDP 指的是当所有核心都工作在基准频率的情况下的平均散热能耗。既然是 Base Frequency 下运行的值,那么在开启 Turbo Boost 后,调整时钟频率后处理器的

用电能耗肯定会增加，处理器的热能耗会不会就此超过 TDP 呢？

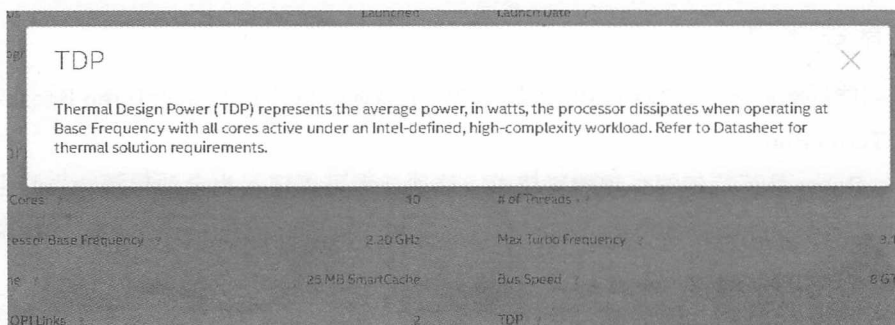


图 5-4 关于 TDP 的解释

Intel 已经告诉我们 TDP 和处理器用电能耗一点关系都没有，TDP 是指处理器的最大热能耗，开启 Turbo Boost 后，处理器的热能耗也不会超过 TDP 的限制。而且 Intel 对于 Turbo Boost 也有专门的说明<sup>①</sup>。

What is Intel® Turbo Boost Technology? Intel® Turbo Boost Technology is a way to automatically run the processor core faster than the noted frequency. The processor must work in the power, temperature, and specification limits of the thermal design power (TDP). Single and multi-threaded application performance increase.

看来，TDP 的确是最大热能耗无疑，而且 Turbo Boost 开启与否都会受到 TDP 的限制。至于那个 Base Frequency，如果不是笔误，我们推测有可能 Turbo Boost 开启后根本就不会增加处理器实际的热能耗。

### 1. 能耗测试的意义

我们这里指的是用电能耗的测试。这个数值在数据中心设计的最初阶段起着相当重要的作用。当我们初步完成设备选型范围的时候，能耗测试会帮助我们完成最终的方案落地。在采购机柜数量的问题上，没有单节点设备的最大能耗值，你就无法决定一个机柜的设备上架数量，也就没法确定最终要购买多少机柜。在海量模式中，每个机柜的设备上架数哪怕只相差一台，商务价格都会出现很大的波动。此外，能耗测试可以剔除掉能耗过高的设备。这种设备将影响原有的设备上架规划，是不能使用的。

### 2. 能耗测试的分类

能耗测试大体可以分为两种。第一种是最大能耗测试，第二种是负载能耗比测试。

最大能耗测试用于获取服务器在最大负载的状态下所消耗的功率。假设测试结果是 220W，那么就是说单台设备所消耗的最大电流是 1A，一个 16A 的机柜减去 1A 的常量值，

① 上述说明出处来自 <http://www.intel.com/content/www/us/en/support/processors/000005641.html?wapkw=turbo+boost>。

可以安置 15 台该型号的服务器，而且 15 是一个绝对安全值，肯定不会发生电力过载的问题。

负载能耗比测试用于评测服务器在不同负载场景下所消耗的功率。我们把它想象成性价比，假设你日常的负载在 70%~80%，那么可以根据这个区间的负载来衡量实际能耗情况。我们用汽车举个例子。有些车的发动机在低转速的时候，显得特别“肉”，可转速一旦超过 3000 转，瞬间又会爆发出很强的动力，而另外一些车却恰恰相反。因此，在选择不同车辆的时候，需要根据车主的驾驶习惯进行挑选。能耗也是一样的道理，不同产品在不同负载区间中的能耗表现也是不一样的，使用负载能耗比测试可以在多个产品之间完成横向对比观察。

### 3. 能耗测试工具

常用的能耗测试工具有如下这些。

- ☐ yes;
- ☐ dd;
- ☐ top;
- ☐ mpstat;
- ☐ sar;
- ☐ turbostat;
- ☐ ipmitool;
- ☐ SPECpower\_ssj2008;
- ☐ 能耗分析器;
- ☐ 温度传感器。

### 4. 能耗测试的要求

在开始能耗测试工作前，我们要明确几个问题。

首先，我们既然是要测量能耗，那么所有限制能耗的选项都应当被禁用，包括 C1E、C State、P State 等。我们可以在 BIOS 里面找到关于 Processor 或者 System 等相关方面的选项，然后进行检查和设置。不要使用厂商的自带模式，一定要通过自定义模式，亲自确认每一个选项。

其次，能耗测试不等于压力测试，我们的目的是让设备全力运转起来，但不是要求性能达到最佳状态，没有必要把心思过多放在内核参数的调整上。从 TDP 的描述我们可以了解到，当设备满载的时候热能耗最高，那么此时的用电能耗也最高。但就整体硬件设计而言，能耗三大件包括 CPU、散热和硬盘。各家硬件的设计理念不尽相同，所以能耗不一定是完全线性的。有些硬件可能在中间的某个阶段就达到了峰值。这个时候，即便再怎么提升负载，但用电能耗就是不见增加。若是一味强行加载，系统反而会因为过载保护去降低能耗的使用。



再次，观察点的选择要正确。我们在进行能耗测试的时候，需要关注的是和能耗有关的参数，例如核心温度、整包功率限制、风扇转速等，不能只盯着资源利用率去看。

最后，能耗测试不能偷懒，这不是一项办公室工作。测量数据需要严谨对待，必须亲临现场去完成测试。另外，能耗测量值读取之前，需要稳定一段时间，待数值没有发生变化的情况下，才能记录数据。

## 5. 使用 turbostat 查询信息

turbostat 是一款读取 Intel 处理器的信息与状态的工具。在实施能耗测试之前，我们可以使用 turbostat -v 来读取拓扑、频率、状态、温度和功耗等相关方面的内容。下面这段代码展示的就是 Intel E5 2650 v2 的详细情况：

```
turbostat v3.2 February 11, 2013 - Len Brown <lenb@kernel.org>
CPUID(0): GenuineIntel 13 CPUID levels; family:model:stepping 0x6:3e:4 (6:62:4)
CPUID(6): APERF, DTS, PTM
RAPL: 690 sec. Joule Counter Range
cpu0: MSR_NHM_PLATFORM_INFO: 0xc10e4811a00
12 * 100 = 1200 MHz max efficiency
26 * 100 = 2600 MHz TSC frequency
cpu0: MSR_IA32_POWER_CTL: 0x25000059 (C1E: DISabled)
cpu0: MSR_IVT_TURBO_RATIO_LIMIT: 0x1e1e1e1e1e1e1e1e
30 * 100 = 3000 MHz max turbo 16 active cores
30 * 100 = 3000 MHz max turbo 15 active cores
30 * 100 = 3000 MHz max turbo 14 active cores
30 * 100 = 3000 MHz max turbo 13 active cores
30 * 100 = 3000 MHz max turbo 12 active cores
30 * 100 = 3000 MHz max turbo 11 active cores
30 * 100 = 3000 MHz max turbo 10 active cores
30 * 100 = 3000 MHz max turbo 9 active cores
cpu0: MSR_NHM_SNB_PKG_CST_CFG_CTL: 0x00008400 (locked: pkg-cstate-limit=0: pc0)
cpu0: MSR_NHM_TURBO_RATIO_LIMIT: 0x1e1e1e1e1e1e1e1e
30 * 100 = 3000 MHz max turbo 8 active cores
30 * 100 = 3000 MHz max turbo 7 active cores
30 * 100 = 3000 MHz max turbo 6 active cores
30 * 100 = 3000 MHz max turbo 5 active cores
31 * 100 = 3100 MHz max turbo 4 active cores
32 * 100 = 3200 MHz max turbo 3 active cores
33 * 100 = 3300 MHz max turbo 2 active cores
34 * 100 = 3400 MHz max turbo 1 active cores
cpu0: MSR_RAPL_POWER_UNIT: 0x000a1003 (0.125000 Watts, 0.000015 Joules, 0.000977 sec.)
cpu0: MSR_PKG_POWER_INFO: 0x2f04b001e002f8 (95 W TDP, RAPL 60 - 150 W, 0.045898 sec.)
cpu0: MSR_PKG_POWER_LIMIT: 0x68390005a82f8 (UNlocked)
cpu0: PKG Limit #1: ENabled (95.000000 Watts, 10.000000 sec, clamp DISabled)
cpu0: PKG Limit #2: ENabled (114.000000 Watts, 0.007812* sec, clamp DISabled)
cpu0: MSR_DRAM_POWER_INFO,: 0x2f0098003e0088 (17 W TDP, RAPL 8 - 19 W, 0.045898 sec.)
cpu0: MSR_DRAM_POWER_LIMIT: 0x00000000 (UNlocked)
cpu0: DRAM Limit: DISabled (0.000000 Watts, 0.000977 sec, clamp DISabled)
cpu0: MSR_PP0_POLICY: 0
```

```

cpu0: MSR_PP0_POWER_LIMIT: 0x00000000 (UNlocked)
cpu0: Cores Limit: DISabled (0.000000 Watts, 0.000977 sec, clamp DISabled)
cpu1: MSR_RAPL_POWER_UNIT: 0x000a1003 (0.125000 Watts, 0.000015 Joules, 0.000977 sec.)
cpu1: MSR_PKG_POWER_INFO: 0x2f04b001e002f8 (95 W TDP, RAPL 60 - 150 W, 0.045898 sec.)
cpu1: MSR_PKG_POWER_LIMIT: 0x68390005a82f8 (UNlocked)
cpu1: PKG Limit #1: ENabled (95.000000 Watts, 10.000000 sec, clamp DISabled)
cpu1: PKG Limit #2: ENabled (114.000000 Watts, 0.007812* sec, clamp DISabled)
cpu1: MSR_DRAM_POWER_INFO,: 0x2f0098003e0088 (17 W TDP, RAPL 8 - 19 W, 0.045898 sec.)
cpu1: MSR_DRAM_POWER_LIMIT: 0x00000000 (UNlocked)
cpu1: DRAM Limit: DISabled (0.000000 Watts, 0.000977 sec, clamp DISabled)
cpu1: MSR_PP0_POLICY: 0
cpu1: MSR_PP0_POWER_LIMIT: 0x00000000 (UNlocked)
cpu1: Cores Limit: DISabled (0.000000 Watts, 0.000977 sec, clamp DISabled)
cpu0: MSR_IA32_TEMPERATURE_TARGET: 0x00590a00 (89 C)
cpu1: MSR_IA32_TEMPERATURE_TARGET: 0x00590a00 (89 C)
cpu0: MSR_IA32_PACKAGE_THERM_STATUS: 0x88200800 (57 C)
cpu0: MSR_IA32_THERM_STATUS: 0x88240800 (53 C +/- 1)
cpu2: MSR_IA32_THERM_STATUS: 0x88260800 (51 C +/- 1)
cpu4: MSR_IA32_THERM_STATUS: 0x88240800 (53 C +/- 1)
cpu6: MSR_IA32_THERM_STATUS: 0x88230800 (54 C +/- 1)
cpu8: MSR_IA32_THERM_STATUS: 0x88230800 (54 C +/- 1)
cpu10: MSR_IA32_THERM_STATUS: 0x88200800 (57 C +/- 1)
cpu12: MSR_IA32_THERM_STATUS: 0x88230800 (54 C +/- 1)
cpu14: MSR_IA32_THERM_STATUS: 0x88200800 (57 C +/- 1)
cpu1: MSR_IA32_PACKAGE_THERM_STATUS: 0x881e0000 (59 C)
cpu1: MSR_IA32_THERM_STATUS: 0x88270000 (50 C +/- 1)
cpu3: MSR_IA32_THERM_STATUS: 0x881e0000 (59 C +/- 1)
cpu5: MSR_IA32_THERM_STATUS: 0x88210000 (56 C +/- 1)
cpu7: MSR_IA32_THERM_STATUS: 0x88220000 (55 C +/- 1)
cpu9: MSR_IA32_THERM_STATUS: 0x881f0000 (58 C +/- 1)
cpu11: MSR_IA32_THERM_STATUS: 0x88210000 (56 C +/- 1)
cpu13: MSR_IA32_THERM_STATUS: 0x88220000 (55 C +/- 1)
cpu15: MSR_IA32_THERM_STATUS: 0x88220000 (55 C +/- 1)

```

通过官方网站的产品查询，大家可以知道 Intel E5 2650 v2 这款处理的主频是 2.6 GHz，核心总数是 8。我们通过 turbostat 的报告可以得知，这台设备上的处理器是双路的，当开启 Turbo Boost 技术，所有核心的主频都会调整到 3 GHz，并且告知我们在什么情况下，可以达到最大的 3.4 GHz。这些描述都和官方网站的描述是一致的。除了静态信息，我们还可以读取一些动态的内容。

```

cpu0: MSR_IA32_POWER_CTL: 0x25000059 (C1E: DISabled)
cpu0: MSR_NHM_SNB_PKG_CST_CFG_CTL: 0x00008400 (locked: pkg-cstate-limit=0: pc0)

```

这两行告诉我们，处理器的 C1E、C State 等功能已经被禁用了，如果发现有类似于 pkg-cstate-limit=3 的这种描述，说明 C State 没有关闭。这个选项在大部分高并发重负载的场景中，都应该采取禁用方式。除非你的生产环境 C State 就是开启的，否则在后续测试工作中所出具的所有结果都是不准确的。

```
cpu0: MSR_IA32_PACKAGE_THERM_STATUS: 0x88200800 (57 C)
cpu0: MSR_IA32_THERM_STATUS: 0x88240800 (53 C +/- 1)
```

第一行指的是整包的热能耗温度 PTMP，cpu0 代表的是第一颗处理器。第二行指的是核心的热能耗温度，cpu0 代表第一个核心。Turbostat 可以展示每个核心的温度。

```
cpu0: MSR_PKG_POWER_INFO: 0x2f04b001e002f8 (95 W TDP, RAPL 60 - 150 W, 0.045898 sec.)
cpu0: PKG Limit #1: ENabled (95.000000 Watts, 10.000000 sec, clamp DISabled)
cpu0: PKG Limit #2: ENabled (114.000000 Watts, 0.007812* sec, clamp DISabled)
```

MSR\_PKG\_POWER\_INFO 反映了这款处理的 TDP 和 RAPL 的值，RAPL 是指运行时的平均功率限制，PKG Limit 代表处理器在单位时间内对功率增长的限制，它的作用是为了防止处理器能耗在短时间内增长过快损坏硬件而设计的。

## 6. 能耗结果读取

我们前面提到过，能耗测试关系到数据中心机柜的采购需求，必须到机房现场才能完成。有些同学可能想通过 ipmitool 工具查看的方式来投机取巧，这种想法是非常错误的。

之所以说 ipmitool 的方式不可取，是因为它借助读取控制芯片上的传感器而获取数据。而不同厂商、不同设备上面的控制芯片中的传感器读数并不能准确反映实际的能耗数值，具体原因和传感器设计有关系。通常传感器只能读取服务器自身能耗，但精准的能耗情况是需要通过在 PDU 上测量才能够得出的。在笔者所接触过的二三十种机型当中，有些设备的传感器读数和最终测量结果相近，而有些则会出现比较大的偏差。即便是同一个品牌，都存在着不同机型得到不同结果的情况。这需要在现场使用专用的能耗分析仪来测量。能耗分析仪的连接方式大同小异，原理就是把能耗分析仪串联到服务器和 PDU 之间，相当于在电源线上接了一块电表来读取电压、电流和能耗等数值。

那么 ipmitool 这个工具能不能用呢？答案是肯定的。ipmitool 可以用，但是不是用于测量数值，ipmitool 是在前期调试的过程中做横向对比分析的。我们在拿到一台全新机型的时候，测试不见得就一帆风顺，开始也许需要多次的调整操作，这个时候可以使用 ipmitool 来观察传感器上面的数据变化，当数据稳定后并确认测试方法准确的时候，再到现场进行实际测量。调试工作的时间是比较长的，而实际测量读取记录数据只需要几分钟。ipmitool 可以让我们在调试阶段实现远程操作，减少人员在机房的驻留时间。

以下是使用 ipmitool 读取温度、电流、风扇转速和能耗的方法。

```
[root@station103 ~]# watch -n1 'ipmitool sdr |egrep -i "Watts|degrees|Amps|RPM|Power|Pwr" |sort'
Every 1.0s: ipmitool sdr|egrep -i "Watts|degrees|Amps|RPM|Power|Pwr"|sort Fri
Mar 17 13:30:21 2017

Current 1          | 0.60 Amps          | ok
Current 2          | 0.80 Amps          | ok
Exhaust Temp       | 38 degrees C       | ok
Fan1A RPM          | 5400 RPM           | ok
```



|                 |              |    |
|-----------------|--------------|----|
| Fan1B RPM       | 5160 RPM     | ok |
| Fan2A RPM       | 5280 RPM     | ok |
| Fan2B RPM       | 5160 RPM     | ok |
| Fan3A RPM       | 5520 RPM     | ok |
| Fan3B RPM       | 5280 RPM     | ok |
| Fan4A RPM       | 5400 RPM     | ok |
| Fan4B RPM       | 5280 RPM     | ok |
| Fan5A RPM       | 5400 RPM     | ok |
| Fan5B RPM       | 5280 RPM     | ok |
| Fan6A RPM       | 5400 RPM     | ok |
| Fan6B RPM       | 5160 RPM     | ok |
| Fan7A RPM       | 5160 RPM     | ok |
| Fan7B RPM       | 5160 RPM     | ok |
| Inlet Temp      | 12 degrees C | ok |
| Power Cable     | 0x01         | ok |
| Power Cable     | Not Readable | ns |
| Power Optimized | 0x01         | ok |
| Pwr Consumption | 294 Watts    | ok |
| Temp            | 70 degrees C | ok |
| Temp            | 70 degrees C | ok |

## 7. 最大能耗测试

我们可以使用 `yes` 和 `dd` 命令完成最大能耗测试的任务。

❑ `yes` 是一个无限循环输出指定字符串的命令，开启 `yes` 后可以很容易把处理器的利用率打满。因为我们要测试的处理器是多核的，因此需要开启与逻辑核心数相对应的 `yes` 的进程数量。默认情况下，处理器是开启超线程的，所以逻辑核心数是物理核心数的两倍。

❑ `dd` 对于读者朋友们来说是非常熟悉的命令了，可以通过 `dd` 读写磁盘来达到让磁盘运转的目的。

关于使用 `yes` 对处理器的测试工作，我们可以通过下面这个简单的脚本来完成。

```
#!/bin/bash

killall yes && sleep 10

KEYWORD="2630|2650"

for i in $(seq 1 `egrep -c "$KEYWORD" /proc/cpuinfo`)
do
    cpu=`expr $i - 1`
    taskset -c $cpu yes &> /dev/null &
done

watch -n1 'ipmitool sdr |egrep -i "Watts|degrees|Amps|RPM" |sort'
```

这个脚本的目的是根据实际开启的逻辑核心数生成数量对等的 `yes` 进程来运行。注意 `KEYWORD` 是一个变量，这里面的 2630 和 2650 是指处理器的型号，读者需要根据自己的

实际情况对关键词进行调整。请注意，运行 yes 进程之前需要进行核心绑定，绑定核心后所有核心才会达到满载。

执行 top 按数字键 1，可以实时观察处理器的利用率。

```
[root@station103 ~]# top
top - 13:31:26 up 24 days, 19:55, 6 users, load average: 31.97, 26.77, 20.31
Tasks: 542 total, 33 running, 509 sleeping, 0 stopped, 0 zombie
Cpu0: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu14: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15: 99.3%us, 0.7%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu17: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu18: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu22: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu23: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu24: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu25: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu26: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu27: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu28: 100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu29: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu30: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu31: 99.7%us, 0.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 65920828k total, 65641064k used, 279764k free, 62741128k buffers
Swap: 33554424k total, 1553944k used, 32000480k free, 36876k cached
```

另一种观测方法是使用 mpstat。

```
[root@station103 ~]# mpstat -P ALL 1
Linux 2.6.32-431.el6.x86_64 (station103.example.com) 03/17/2017 _x86_64_ (32 CPU)

01:32:04 PM CPU %usr %nice %sys %iowait %irq %soft %steal %guest %idle
01:32:05 PM all 99.75 0.00 0.25 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 0 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

```

01:32:05 PM 1 99.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 2 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 3 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 4 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 5 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 6 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 7 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 8 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 9 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 10 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 11 99.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 12 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 13 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 14 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 15 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 16 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 17 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 18 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 19 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 20 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 21 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 22 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 23 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 24 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 25 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 26 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 27 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 28 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 29 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 30 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
01:32:05 PM 31 99.01 0.00 0.99 0.00 0.00 0.00 0.00 0.00 0.00 0.00

```

当处理器满载后，我们打开 `turbostat -S` 观察如下这些参数。

- %c0: 利用率;
- GHz: 实际频率;
- TSC: 设定频率;
- CTMP: 核心温度;
- PTMP: 整体温度;
- Pkg\_W: 整体能耗;
- Cor\_W: 核心能耗;
- RAM\_W: RAM 能耗。

```

[root@station103 ~]# turbostat -S
    %c0  GHz  TSC  SMI    %c1    %c3    %c6    %c7  CTMP  PTMP    %pc2    %pc3    %pc6
%pc7  Pkg_W  Cor_W  RAM_W  PKG_%  RAM_%
    0.02  3.00  2.60   0   99.98   0.00   0.00   0.00   57   57   0.00   0.00   0.00   0.00
80.15  53.75  8.72   0.00   0.00

```



```

      0.01 3.00 2.60 0 99.99 0.00 0.00 0.00 56 57 0.00 0.00 0.00 0.00
80.07 53.69 8.72 0.00 0.00
      28.39 3.00 2.60 0 71.61 0.00 0.00 0.00 66 66 0.00 0.00 0.00 0.00
100.40 73.97 8.72 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 68 68 0.00 0.00 0.00 0.00
152.48 125.74 8.72 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 69 69 0.00 0.00 0.00 0.00
153.11 126.19 8.73 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 69 69 0.00 0.00 0.00 0.00
153.30 126.29 8.71 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 70 70 0.00 0.00 0.00 0.00
153.68 126.62 8.66 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 70 70 0.00 0.00 0.00 0.00
154.02 126.92 8.69 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 71 71 0.00 0.00 0.00 0.00
154.19 127.07 8.68 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 72 71 0.00 0.00 0.00 0.00
154.38 127.25 8.66 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 72 71 0.00 0.00 0.00 0.00
154.59 127.43 8.67 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 71 71 0.00 0.00 0.00 0.00
154.60 127.43 8.71 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 71 71 0.00 0.00 0.00 0.00
154.61 127.41 8.69 0.00 0.00
      100.00 3.00 2.60 0 0.00 0.00 0.00 0.00 71 71 0.00 0.00 0.00 0.00
154.57 127.36 8.71 0.00 0.00

```

我们看到因为开启了 Turbo Boost 的缘故，2650 v2 的实际工作频率是高于标称值的。

有意思的是，如果没有进行核心绑定，idle 还会有比较大的剩余，不过此时从 ipmitool 上得到的能耗读数和绑定核心处理器资源满载的情况基本上是一致的。在不绑定核心的情况下，我们重复运行两次，使得 yes 的进程数为超线程核心数的两倍，此时 idle 为 0，但是能耗几乎没有增长。通过 turbostat 和 ipmitool 命令进行观察可以发现，虽然处理器的利用率提升了，但是温度没有变化，都是 71℃ 左右，风扇转速和电流读数也没有什么变化。那么，能耗没有明显增长也就不奇怪了。实验也说明了性能与能耗之间没有绝对的联系，两者之间并非是完全线性的，能耗是呈阶梯式上升的。有兴趣的读者可以自行测试。

处理器打满后，我们再利用 dd 驱动磁盘全速运行。这里不同的产品会出现差异，有些服务器在加入 dd 后能耗没有任何增加，而有些服务器的能耗则有少量提升。dd 的基本命令如下所示。

```
# dd if=/dev/sdX of=/dev/null
```

如果是多块磁盘，需要确认所有磁盘都运转起来了。dd 同样需要绑定 CPU 核心，否则在多块磁盘测试的过程中 I/O 会在各个磁盘之间乱串，达不到最佳效果。我们可以使用 iostat 或者 sar 来确认运行的效果。

```
avg-cpu:  %user %nice %system %iowait  %steal   %idle
           92.25  0.00   7.75   0.00   0.00   0.00
```

| Device: | tps     | Blk_read/s | Blk_wrtn/s | Blk_read | Blk_wrtn |
|---------|---------|------------|------------|----------|----------|
| sda     | 579.00  | 145920.00  | 0.00       | 145920   | 0        |
| sdb     | 946.00  | 241152.00  | 0.00       | 241152   | 0        |
| sdc     | 1079.00 | 274688.00  | 0.00       | 274688   | 0        |
| sdd     | 978.00  | 249344.00  | 0.00       | 249344   | 0        |
| sde     | 976.00  | 248320.00  | 0.00       | 248320   | 0        |
| sdf     | 1094.00 | 278784.00  | 0.00       | 278784   | 0        |
| sdg     | 1407.00 | 357120.00  | 0.00       | 357120   | 0        |
| sdh     | 672.00  | 171008.00  | 0.00       | 171008   | 0        |
| sdi     | 678.00  | 171624.00  | 0.00       | 171624   | 0        |
| sdj     | 597.00  | 151552.00  | 0.00       | 151552   | 0        |
| sdk     | 596.00  | 151296.00  | 0.00       | 151296   | 0        |
| sdl     | 565.00  | 142744.00  | 0.00       | 142744   | 0        |

### 5.3.5 CPU 性能测试

sysbench 是一款模块化、多线程的用于基准测试的性能评估工具。它可以在没有复杂环境配置的情况下，快速得到一个关于系统性能的评价，它可以用于 CPU、内存、文件 I/O、POSIX 线程调度、OLTP 等一些类的性能测试。原本 sysbench 主要就是为 MySQL 服务器的基准测试所写，而后又被进一步扩展到支持数据库多实例、分布式基准和第三方插件模块。

在本节我们主要用于测试 CPU 的性能。使用如下命令进行测试。

```
# sysbench --num-threads=32 --max-requests=10000 \
--test=cpu --cpu-max-prime=10000 run
```

其中：

- ❑ `--num-threads` 全局选项，用于设置线程数量。具体数字可以用命令 `grep -c KEYWORD /proc/cpuinfo` 查看，如果要测试单核心，需要使用命令 `echo 0 > /sys/devices/system/cpu/cpuX/online` 来关闭其余的核心。
- ❑ `--max-requests` 全局选项，用于设置请求次数。这个数值理论上会被启用的线程平分处理。
- ❑ `--test` 测试模块选择，`--test=cpu` 代表本次测试为 CPU 测试环节。
- ❑ `--cpu-max-prime` 是 CPU 测试子模块选项，用于设置求解范围在 N 以内的素数。

命令运行后所得结果如下。

```
[root@station101 ~]# time sysbench --num-threads=32 --max-requests=10000
--test=cpu --cpu-max-prime=30000 run
sysbench 0.4.12: multi-threaded system evaluation benchmark
```

Running the test with following options:

```

Number of threads: 32
Doing CPU performance benchmark
Threads started!
Done.

Maximum prime number checked in CPU test: 30000

Test execution summary:
    total time:                2.3121s
    total number of events:    10000
    total time taken by event execution: 73.2848
    per-request statistics:
        min:                    4.50ms
        avg:                    7.33ms
        max:                    125.61ms
        approx. 95 percentile: 15.61ms

Threads fairness:
    events (avg/stddev):       312.5000/79.31
    execution time (avg/stddev): 2.2902/0.02

real    0m2.315s
user    0m53.503s
sys     0m0.005s

```

- total time 指的是程序运行时间。
- total time taken by event execution 是指所有事件花费的时间。正常情况下，这个时间应当接近 total time 与 --num-threads 的乘积。total time 并不是一个核心所花费的时间，但由于任务会尽可能平均分布在所有的核心上面，所以 total time 与 execution time 的平均值应该比较接近。
- per-request statistics 表示所有请求执行时间的情况，95 percentile 代表 95% 的样本均值。本例中说明绝大多数（95%）的 events 执行时间为 15.61ms。
- Threads fairness 体现单个核心负载均衡性的情况，events 的平均值就是 --max-requests 除以 --num-threads 后，理论应当分配的 events 数量。execution time 的平均值就是单个核心的平均执行时间。stddev 是标准差，体现了实际测试结果和平均值之间的偏离值。这个数值越大，表示测试所得结果的随机性就越强，测试可信度就越低。标准差的计算公式为。

$$\text{stddev} = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

### 5.3.6 内存性能测试

内存的测试我们依旧使用 sysbench 完成，使用 --test=memory 展开内存测试。memory 子模块的相关参数如下。



- ❑ `--memory-block-size`: 设置内存块的大小, 默认为 1KB。
- ❑ `--memory-total-size`: 设置内存大小, 按照你的硬件实际配置情况设定。
- ❑ `--memory-scope`: 可以设定 `global` 和 `local`, 默认为 `global`, 开启 `numa` 的情况下要修改成 `local`。
- ❑ `--memory-hugetlb`: 设定 HugeTLB, 默认为关闭。
- ❑ `--memory-oper`: 设定读写方式, 包括 `read`、`write` 和 `none`, 默认为 `write`。
- ❑ `--memory-access-mode`: 设定访问模式, 包含顺序和随机, 默认为顺序。

`--test=memory` 的输出结果格式与 `--test=cpu` 是一样的, 具体数据分析不再赘述。详细内容请参见 5.3.5 节。

### 5.3.7 磁盘性能测试

我们前面在选型的过程中, 是依照磁盘的参数来选择, 但是磁盘是要完成 RAID 后才能给操作系统使用的。此时磁盘自身的参数已经不能说明太多问题了, 需要进行实际的磁盘性能测试。

虽然市场上的 RAID 卡大多是基于 LSI 进行设计的, 但是各大厂商并不会完全奉行拿来主义, 每款产品或多或少都会有些变化, 因此它们相互之间必然会有一些差异。如果各家厂商使用的是同一款硬盘, 那么性能差异就会直接体现在 RAID 卡上面。

在测试磁盘性能之前, 应当先清除缓存, 确保测试结果的公正性。

要达到释放缓存的目的, 涉及配置文件 `/proc/sys/vm/drop_caches`。这个文件中记录了缓存释放的参数。这个参数的取值范围是 0~3, 默认值为 0 (即不释放缓存), 其他数值的解释如下。

- ❑ `drop_caches = 1` - 释放 `pagecache`;
- ❑ `drop_caches = 2` - 释放 `dentry` 和 `inode`;
- ❑ `drop_caches = 3` - 释放 `pagecache`、`dentry` 和 `inode`。

这个操作没有涉及 Dirty Pages 的内容, 并不会触发写回操作。为了彻底清除干净, 应当先进行 `sync` 刷新缓存后, 再使用命令 `echo 3 > /proc/sys/vm/drop_caches` 来释放缓存。该指令输入后会立即生效。

而当我们配合 `free` 命令时, 可以清楚地看到整个清空过程中缓存的变化。

```
[root@station101 ~]# cat /proc/sys/vm/drop_caches
0
[root@station101 ~]# free -m
      Total    used    free      shared    buffers     cached
Mem:  64375  780    63595         0         40       143
-/+ buffers/cache: 596    63779
Swap:      32767         0       32767
[root@station101 ~]# echo 2 > /proc/sys/vm/drop_caches
```

```
[root@station101 ~]# free -m
      Total        used       free      shared    buffers     cached
Mem: 64375         624       63751          0         40         17
-/+ buffers/cache: 566       63808
Swap: 32767          0       32767

[root@station101 ~]# echo 1 > /proc/sys/vm/drop_caches
[root@station101 ~]# free -m
      Total        used       free      shared    buffers     cached
Mem: 64375         576       63798          0          0         10
-/+ buffers/cache: 565       63810
Swap: 32767          0       32767
```

因为测试过程中需要创建大量的文件和进程，因此需要解除 ulimit 的限制，建议打开如下参数。

```
ulimit -d unlimited
ulimit -f unlimited
ulimit -i unlimited
ulimit -m unlimited
ulimit -s unlimited
ulimit -n 1000000
ulimit -l 67108864
```

此外，还要根据测试需求，调整缓存的写入策略和系统调度算法。例如，如果你要测试 SSD，建议将调度算法设置成 noop 模式，而 HDD 可以设置为 deadline 或者 cfq。注意：这些调整必须和实际生产环境中的设置一致。

```
# MegaCli -LDGetProp -Cache -L0 -aALL // 检查缓存策略是否开启
# MegaCli -LDSetProp WT|WB|NORA|RA|ADRA -L0 -aALL // 设置缓存策略
# echo "deadline" > /sys/block/sdX/queue/schedule // 调整 HDD 调度算法
# echo "noop" > /sys/block/sdX/queue/schedule // 调整 SSD 调度算法
```

测试内容应当根据业务实际需要，在基于顺序、随机等不同的读写场景，进行详尽的测试。表 5-2 所示为一个常用示例，但 Block Size 不应该拘泥于本表提供的数值，需根据实际的业务情况进行调整。例如，你的生产环境是视频站点，那么实际系统 Block Size 数值的设置就要比本表大得多。

表 5-2 磁盘性能测试范例表

|        | Seq R | Seq W | Seq RW | Rnd R | Rnd W | Rnd RW |
|--------|-------|-------|--------|-------|-------|--------|
| bs=1K  |       |       |        |       |       |        |
| bs=2K  |       |       |        |       |       |        |
| bs=4K  |       |       |        |       |       |        |
| bs=8K  |       |       |        |       |       |        |
| bs=16K |       |       |        |       |       |        |
| bs=32K |       |       |        |       |       |        |

(续)

|          | Seq R | Seq W | Seq RW | Rnd R | Rnd W | Rnd RW |
|----------|-------|-------|--------|-------|-------|--------|
| bs=64K   |       |       |        |       |       |        |
| bs=128K  |       |       |        |       |       |        |
| bs=256K  |       |       |        |       |       |        |
| bs=512K  |       |       |        |       |       |        |
| bs=1024K |       |       |        |       |       |        |

### 1. 基于 sysbench 的测试

sysbench 使用 `--test=fileio` 展开 I/O 测试。整个测试分成三步：创建测试文件，执行测试和清理测试文件。fileio 子模块的相关参数如下。

- ☐ `--file-num`: 设定创建文件的数量，默认为 128 个；
- ☐ `--file-block-size`: 设定 Block Size，默认为 16384；
- ☐ `--file-total-size`: 设定文件总计大小，默认为 2GiB；
- ☐ `--file-test-mode`: 设定模式，包括 seqwr、seqrewr、seqrd、rndrd、rndwr 和 rndrw 这六种；
- ☐ `--file-io-mode`: 设定 I/O 模式，包括 sync、async、fastmmap、slowmmap，默认为 sync；
- ☐ `--file-async-backlog`: 设定每个线程队列的异步操作数，默认为 128；
- ☐ `--file-extra-flags`: 设定 open() 的 flag，包括 sync、dsync、direct；
- ☐ `--file-fsync-freq`: 设定多少个请求后执行 fsync()，如果设定为 0 则不会执行同步，默认为 100；
- ☐ `--file-fsync-all`: 每一个写操作完成后都要执行一次 fsync()，默认为禁用；
- ☐ `--file-fsync-end`: 设定测试结束时执行 fsync()，默认为启用；
- ☐ `--file-fsync-mode`: 设定同步方法，包括 fsync、fdatasync，默认为 fsync；
- ☐ `--file-merged-requests`: 设定可以合并 I/O 的请求数，默认是 0 执行合并操作；
- ☐ `--file-rw-ratio`: 设定读写比例，用于混合读写的测试，默认为 1.5，也就是 3:2，即 60% 读，40% 写。

sysbench 的执行范例如下。

```
sysbench --test=fileio --file-num=1024 --file-total-size=1G prepare
sysbench --test=fileio \
--max-time=1200 --max-requests=100000 --num-threads=32 \
--init-rng=on --file-test-mode=rndrd \
--file-num=1024 --file-total-size=1G --file-block-size=16384 \
--file-extra-flags=direct --file-fsync-freq=0 run
sysbench --test=fileio --file-num=1024 --file-total-size=1G cleanup
```

### 2. 基于 fio 的测试

fio 是一个非常灵活的测试工具，它通过多线程或进程模拟各种 I/O 操作。fio 可以将最



最终的测试结果输出到 cvs 文件当中，将 cvs 导入 Excel 可以产出最终的性能数据报告。fio 的相关参数包括全局参数和 JOB 参数。

常用全局参数如下。

- ❑ `--output`: 设定输出文件;
- ❑ `--runtime`: 设定运行时间，如果测试是一个 Endless 的场景，需要使用 `-t` 来限定结束;
- ❑ `--latency-log`: 生成延迟日志，日志将包含时间轴和延迟信息，如果使用这个选项，后期可以利用 `fio_generate_plots` 工具绘制数据图;
- ❑ `--bandwidth-log`: 生成带宽日志，日志将包含时间轴和带宽信息，如果使用这个选项，后期可以利用 `fio_generate_plots` 工具绘制数据图;
- ❑ `--minimal`: 最小输出，减少在终端屏幕上的标准输出信息。

常见 JOB 参数如下。

- ❑ `--name`: 设定测试名称;
- ❑ `--numjobs`: 设定开启的线程数;
- ❑ `--norandommap`: 通常 fio 使用随机 I/O 覆盖 Block，它利用 `randommap` 来刻意实现随机操作，每产生一个新 I/O 时，fio 会参考历史 I/O 已确定新 I/O 的偏移位置，确定这个参数后，历史将不再被用于参考，这样做会更符合实际情况一些;
- ❑ `--ramp_time`: 设定预热时间，在正式记录数据之前，应当先等待一段时间，让磁盘进入测试状态;
- ❑ `--rw`: 设定读写模式，包括 `read`、`write`、`rw`、`randread`、`randwrite`、`randrw`;
- ❑ `--size`: 设定本次 I/O 测试的总计大小，当这个条件先于 `--runtime` 达成时，测试将提前结束;
- ❑ `--direct`: 设定是否使用 buffer，默认是 0，即启用 buffer;
- ❑ `--ioengine`: 设定 I/O 如何工作;
- ❑ `--rwmixread`: 设定读写混合时，读操作所占的百分比;
- ❑ `--bs`: 设定 Block Size;
- ❑ `--iodepth`: 设定 I/O 深度，也就是一次可以提交 I/O 的数量;
- ❑ `--group_reporting`: 按照组而非 job 来生成报告。

fio 的执行范例如下。

```
fio --name=myfio_testing \
    --numjob=1 --rw= randrw --rwmixread=70 \
    --ramp_time=30 --runtime=120 --norandommap \
    --bs=16k --iodepth=32 --size=1G \
    --latency-log --bandwidth-log --group_reporting --minimal
```

### 5.3.8 网络性能测试

网络测试是建立在 Client/Server 的方式上完成的。Server 端用来侦听来自 Client 端的连接，Client 端向 Server 端发起网络测试，两端之间会建立起两套通道，一个通道用于传递有关测试的全部信息，另一个通道用于传递数据流量。

#### 1. 基于 iperf 的测试

iperf 是一个网络性能测试工具。它可以测试 TCP 和 UDP 带宽质量，延迟抖动和数据包丢失等情况。

1) iperf 常见全局参数如下。

- ❑ -f: 指定输出单位;
- ❑ -i: 指定时间间隔;
- ❑ -l: 指定读写 buffer 的大小，默认为 8KiB;
- ❑ -m: 打印 MSS (Maximum Segment Size) 信息;
- ❑ -M: 指定 MSS (Maximum Segment Size);
- ❑ -N: 设置 TCP\_NODELAY;
- ❑ -o: 指定输出文件;
- ❑ -p: 指定端口;
- ❑ -u: 使用 UDP 协议，如没有指定该参数则为 TCP 协议;
- ❑ -w: 指定窗口大小。

2) iperf 常见 Server 端参数如下。

- ❑ -s: 以 Server 端的模式运行;
- ❑ -D: 让 Server 端运行在后台;
- ❑ -U: 以单线程 UDP 模式运行。

3) iperf 常见 Client 端参数如下。

- ❑ -c: 以 Client 端的模式运行，-c 后面要指定 Server 端的主机名或 IP 地址;
- ❑ -b: 设定 UDP 的传输带宽，如果测试基于 TCP 协议请不要设定这个参数;
- ❑ -k: 设定传输数据包个数，当设定条件先于 -t 达成时，测试将提前结束;
- ❑ -n: 设定传输字节，当设定条件先于 -t 达成时，测试将提前结束;
- ❑ -O 设定忽略前几秒的数据，用于测试前的就绪准备，仅 iperf3 支持;
- ❑ -P: 设定 Client 端并发数量;
- ❑ -R: 设定反向测试，数据是从 Server 端发起，流向到 Client 端;
- ❑ -S: 设定服务类型;
- ❑ -t: 设定测试时间，如果测试是一个 Endless 的场景，需要使用 -t 来限定结束;
- ❑ -T: 设定输出行的字符串前缀;



- ❑ -Z: 设定使用零拷贝模式发送数据;
- ❑ -4: 使用 IPv4;
- ❑ -6: 使用 IPv6;
- ❑ --get-server-output: 从 Server 端获取结果。

iperf 的执行范例如下, 它的测试结果简单易读, 我们不做过多展示。

```
iperf3 -s -p 50000 -D // 运行 Server 端
iperf3 -c <SERVER_IP> -p 50000 -t 1200 -w 128k // 测试单线程
iperf3 -c <SERVER_IP> -p 50000 -t 1200 -w 128k -P 10 // 测试多线程并发
iperf3 -c <SERVER_IP> -p 50000 -t 1200 -w 128k -u -b 10m // 测试 UDP
iperf3 -c <SERVER_IP> -p 50000 -t 1200 -w 128k -R // 测试反向传输
```

## 2. 基于 netperf 的测试

netperf 可以实施不同模式的网络性能测试。软件可以直接去 netperf 的官网 <ftp://ftp.netperf.org/netperf/> 下载。

编译安装完成后, 我们需要在 Server 端上运行 netserver, 然后在 Client 端上发起测试。

### (1) netserver 常见的全局参数

- ❑ -d: 增加 Debug 输出;
- ❑ -N: 禁用 Debug 输出;
- ❑ -p: portnum 指定监听端口;
- ❑ -4: 使用 IPv4;
- ❑ -6: 使用 IPv6。

### (2) netperf 常见的全局参数

- ❑ -c [cpu\_rate]: 报告本地 CPU 的利用率;
- ❑ -C [cpu\_rate]: 报告对端 CPU 的利用率;
- ❑ -f G|M|K|g|m|k: 指定输出单位, 默认是 m, 为了可读性强, 可以调整成 M, 如果测试万兆网络则可以修改成 G;
- ❑ -H name|ip,fam: 指定目标向谁发起测试;
- ❑ -l testlen: 设定测试时长, 如果测试是一个 Endless 的场景, 需要使用 -l 来限定结束;
- ❑ -n numcpu: 设定处理器数量, 默认 netperf 是使用所有活跃的 CPU 的, 在你只想测试部分 CPU 的情况下, 可以用这个选项覆盖;
- ❑ -p port,lport: 指定 netserver 的端口或者本地端口;
- ❑ -s seconds: 设定测试之前需要等待的时间;
- ❑ -T lcpu,rcpu: 将 Client 端和 Server 端绑定到本地和对端的 CPU 上;
- ❑ -t testname: 设定测试类型, 包括 TCP\_STREAM、UDP\_STREAM、TCP\_RR、TCP\_CRR、UDP\_RR 等;





- ❑ `-v level`: 设定显示信息的级别, 数值越大, 信息越多;
- ❑ `-- -h`: 显示测试参数的帮助信息, 前提是需要指定 `-t testname`。

### (3) netperf 常见的子项参数

- ❑ `-b number`: 指定 `_RR` 系列测试开始时发送的请求数;
- ❑ `-c`: 为 `TCP_CRR` 和 `TCP_CC` 明确声明这是一个测试连接;
- ❑ `-C`: 设置 `TCP_CORK`;
- ❑ `-D [L][,R]`: 为本地或对端设置 `TCP_NODELAY`;
- ❑ `-k [file]`: 生成 Key-Value 文件;
- ❑ `-K loc[,rem]`: 指定本地或对端的平台使用拥塞控制算法;
- ❑ `-m size`: 指定本地系统发送测试分组的大小;
- ❑ `-M size`: 指定对端系统接收测试分组的大小;
- ❑ `-o [file]`: 生成 csv 文件;
- ❑ `-O [file]`: 生成 Classic-Style 文件;
- ❑ `-p min[,max]`: 为 `TCP_CRR`、`TCP_TRR` 指定最小 / 最大端口号;
- ❑ `-r req,[rsp]`: 指定请求和响应的大小;
- ❑ `-s size`: 指定本地系统的 socket 发送与接收缓冲大小;
- ❑ `-S size`: 指定对端系统的 socket 发送与接收缓冲大小。

netperf 的执行范例如下, 它的测试结果简单易读, 我们不做过多展示。

```
netperf -t <TESTNAME> -c -C -f M -H <SERVER_IP> -l 1200 -- -s 128k -S 128K
```

### (4) 常见的 TESTNAME 测试类型

- ❑ `TCP_STREAM`: 基于 TCP 协议, 是 netperf 的默认测试项;
- ❑ `UDP_STREAM`: 基于 UDP 协议;
- ❑ `TCP_MAERTS`: 是 `TCP_STREAM` 的反向测试, 我们注意到拼写就是反着的, 数据是从 Server 端发起, 流向到 Client 端;
- ❑ `UDP_MAERTS`: 是 `UDP_STREAM` 的反向测试;
- ❑ `TCP_SENDFILE`: 类似于 `TCP_STREAM`, 不同的是, netperf 调用 `sendfile()` 来取代 `send()`, 使用零拷贝模式发送数据;
- ❑ `TCP_RR`: 基于 TCP 协议的请求 / 响应测试;
- ❑ `UDP_RR`: 基于 UDP 协议的请求 / 响应测试;
- ❑ `TCP_CC`: 基于 TCP 协议的连接 / 关闭测试;
- ❑ `TCP_CRR`: 基于 TCP 协议的连接 / 请求 / 响应测试;
- ❑ `OMNI`: 基于 OMNI 的测试。



## 扩展知识：什么是零拷贝？

标准 I/O 接口是基于数据拷贝操作的，数据是在内核地址空间的缓冲区和应用程序地址空间定义的缓冲区之间进行传输。一次系统调用至少要经历两次上下文切换和拷贝操作，从而引发较大的 CPU 开销，限制了数据传输操作的能力。

零拷贝（zero-copy）可以有效地改善数据传输的性能，在内核驱动程序（比如网络堆栈或者磁盘存储驱动程序）处理 I/O 数据的时候，零拷贝技术可以在某种程度上消除不必要的拷贝操作。零拷贝流程如图 5-5 所示。

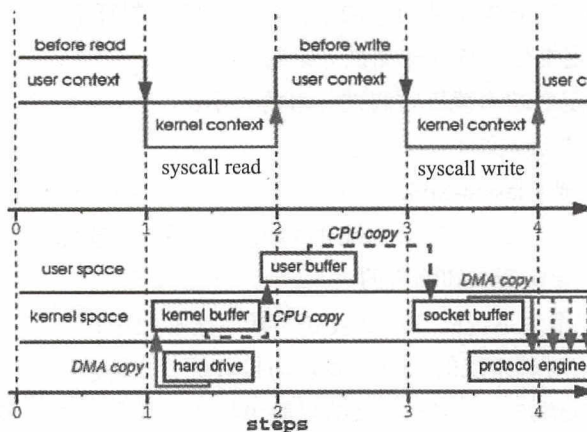


图 5-5 零拷贝流程

### 5.3.9 测试后的收尾工作

测试工作完成后，还需要一些必要的收尾工作，整个测试流程才算结束。

- ☐ 出具服务器产品功能测试报告；
- ☐ 出具服务器性能测试报告；
- ☐ 清除全部的测试数据以及敏感信息；
- ☐ 和相关干系人反馈交流。

严格意义上讲，我们在测试部署系统的时候，应当分离生产环境的数据，至少要把系统密码分离开。密码是很重要的，最好测试的时候设置的密码和生产分开为佳。完成测试后，应当清除敏感数据，包括系统密码、主机名、IP 地址、配置文件、历史命令，还有测试的数据，都不应该留在系统上面。

理论上，修改这些文件七遍以上就可以了。我们可以使用 `shred` 来擦除关键的敏感数据，但是擦出之前，需要关闭 `journal` 日志。可以使用如下命令。

```
# tune2fs -O ^has_journal /dev/sdd1
```

我们擦除数据的顺序应当是先擦除数据文件，后擦除系统文件，在擦除敏感数据的过程中会修改相应的配置文件，因此这个时候要确保连接不能中断，以免任务未完成就无法登录系统。shred 适合清除特定的文件，对于数据盘，使用 dd 是比较简单的办法。

在写本书时，恰好拿到了一台测试设备，在登录带外管理界面时，看到上面赫然留有一家公司的主机名 FQDN 和 DNS 地址配置。

另外，要提醒大家一点的是，测试结果出来后，注意不要对外发布全部的比较结果，尤其是涉及产品的一些具体问题，谁的问题和谁去沟通。不要向 A 厂商提起 B 厂商的产品问题。

## 5.4 本章小结

本章讲述了硬件选型、服务器产品测试和性能测试相关方面的工作要点。硬件选型和性能测试，要选用科学准确的测量方法获取结果。对于产品设计而言，我认为还是要遵从用户为本的原则。论技术功底，可能是厂商更加深厚一些。但是论产品设计，用户才是真正的专家。





## 第 6 章

# 构建 CMDB 与 Workflow

在撰写本章的时候，我偶然间看到了陈皓（左耳朵耗子）先生的一篇杂文——《拖累开发团队效率的困局与解决之道》<sup>①</sup>。文章言语犀利、有话直说的风格是我所喜欢的。其实运维和开发之间有很多地方都是相通的。文中所论之痛点，令同道中人会心一笑。尤其是那犀利的批判性言论，待到戳中问题要害之处，却也有一番快意恩仇的感觉。

本章将是一个独立的篇章，为大家介绍运维平台中最为重要的两大系统 CMDB 和 Workflow。本章旨在谈论构建这两个系统的重要性，以及在构建过程中需要考虑的一些重要因素，并且通过一些故事或实际案例为读者分析构建过程中的一些误区。

当然，在剖析分析问题之前，我们还是老规矩——先讲两个小故事做“药引子”。

---

### 运维故事 9：审计总动员

今年是 Foo 公司创业的第三年。由于业务发展迅速，数据中心的服务器数量已经由原来的 300 台增长到了现在的 5000 台。期间，公司也发生了很多变化，组织架构调整了多次，人员变动也比较频繁。这不，今年刚刚走马上任的审计主管找到审计员小胖，要他负责检查公司的资产情况。结果这一查，却惹出了很多麻烦。

小胖到现场一检查就发现了问题——有 200 台服务器和资产提交的信息列表对不上。资产信息列表是在服务器入库时建立的，当时记录在案的信息都是正常的。而且，资产管理员茜茜很负责，每次设备入库时，她都对入库设备的信息进行核对和记录。然而，小胖在这次检查的过程中发现，资产信息列表上标明的部分主机现在已经不在当初记录的位置上了。出现这个问题是由于设备迁移导致的。迁移工作由业务需求方发起，每次迁移工作的内容都是通过电子邮件来传递，相关干系人也不是固定的。当迁移工作完成后，有些比较负责的人会把迁移信息通过邮件抄送给茜茜，让她完成信息的同步更新。但也有一些人没有这么做，所以导致一些设备做了迁移，但资产管理员这边根本不知道。

---

<sup>①</sup> 感兴趣的读者可以到 <https://coolshell.cn/articles/11656.html> 阅读。



经核查，数据中心服务器的总数是对的。但是这种结论不符合审计的要求。所以，现在所有被查出问题的服务器，都必须一台一台地核对清楚。由于涉及服务器的查找，茜茜先联系了 SE 协助核查。SE 根据问题地址进行核对时发现，有些地址根本 ping 不通。SE 联系了监控团队帮助确认，监控团队说这些问题地址已经不在线上运行了。于是，SE 又去找 PE 帮忙查询。PE 说这些问题地址的系统都已经下线了，服务器也退还了。至于 SN 号，他们从来没有记录过，所以不知道具体是哪台服务器下线了。没办法，SE 只好抓取了所有能登录的服务器的 SN 号出来，整理了一张全新的数据表给茜茜。但是根据 SN 号，SE 也只能找到新的业务 IP 地址和管理地址而已，并不能据此知晓服务器的具体位置。查询历史电子邮件也很困难，有些邮件随着人员离职早就不知道哪里去了。没办法，茜茜还得联系 IDC 的人，通过带外管理地址（好在服务器都有可以看到带外管理地址的液晶显示屏）去核对机架的信息。一项审计工作，前后牵动了审计、资产、监控、SE、PE、IDC 六方共计十多个人。

## 6.1 谁拖了运维的后腿

故事讲完了，我们来看问题出在哪里？

首先，对于数据信息的存储和维护，完全依赖于 Excel 表格，而不是建立在数据库之上。

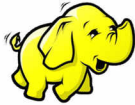
其次，业务变更通过邮件发起，涉及的数据变更，是否会通知茜茜进行同步更新，全凭发起人的高度自觉性。没有数据库支持、没有标准规范、工作流程混乱、缺乏约束条件等各种因素导致了最终的悲剧。

虽然，这只是一个虚拟的故事。但是，这种场景对于读者朋友们来说，想必并不陌生，大家或多或少地都能感觉到一丝熟悉的味道。实际上，类似的问题一直在我们身边不断地发生着。

在企业生产经营当中，最为重要的就是数据信息。它为管理者在决策过程中提供了重要的参考依据，而数据缺失和数据失准又会给决策带来巨大的风险。管理者对数据的价值与重要性非常清楚，但是从实际执行的表现来看，却并不尽如人意。就像一个人购买了一辆新车，车主很爱惜地贴了车膜，还购置了脚垫、座椅套等一大堆配件回来。可是，车却一直停在车库里，既不拿来开，也不去保养。等到真正需要用车的时候却发现——车，开不了了。

传统的 IT 办公系统多是面向单一业务的，没有过多考虑业务之间的联动关系。报销系统就只负责报销，打印系统就只负责打印。然而不幸的是，很多业务处理恰恰不是孤立的。一个业务的执行往往与其他部门、其他人或者其他事务有着很多关联，这就构成了一个复杂的业务流程。由于各个 IT 系统都是孤立的，当业务从一个环节流转到另外一个环节的时





候，发现在 IT 系统层面上根本无路可走。业务流程在这种缺乏有效管理的情况下，各个系统之间的信息交换不得不依靠人工离线来完成。而在这个过程中，产生了大量的邮件、表格、会议和沟通等游离于系统之外的副产品，使得工作效率大打折扣。

### 运维故事 10：丛林野人

话说在一片古老而广阔的丛林中住着一群野人。在这片丛林最深处有一口清泉，这是丛林中唯一的饮水场所。幸运的是，他们居住的洞穴恰好位于这口清泉的旁边。喝水是不成问题的，野人们还需要填饱肚子。好在丛林里到处都长满了可以食用的野果，野人们就靠采集这些野果来充饥。

日子过得很幸福，野人族群渐渐兴旺发达了起来。慢慢地，大家发现，洞穴周边的野果都被吃光了。于是他们不得不到更远的地方去找寻食物。随着时间的推移，探索的路越走越远，很多出去搜寻食物的人再也没有回来。倒不是他们贪玩迷失了方向，而是因为路途太远又没有水喝而渴死了。这就陷入了一个两难的境地，附近没有食物，远方又没有水。这该如何是好呢？身强力壮的野人们开始加强身体的锻炼，希望借此可以让自己走得更远一些。只有一个聪明的野人与众不同，他在空闲的日子里，借助原始工具去尝试着做一个木制的水壶。

又过了些日子，终于方圆五十里内的野果都采光了，这已是野人们体力的极限了。这时候，那个聪明的野人已经成功地完成了自己的作品。他背上一壶水，踏上了探索的征途，当徒步走了三天之后，他惊讶地发现，自己已经站在了丛林的边缘。而不远处是一片新的天地，那里有着丰富的食物和水。原来这片丛林并不像传说中的那么广阔，但是没人能在缺水的情况下在丛林里熬过三天。

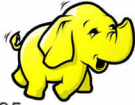
运维的困境就在于，没有提前做好面对质变的准备。团队疲于应付基于故障处理、基于业务需求的工作，完全没有时间，甚至没有就产品价值和用户价值去思考问题的意识。一个开发项目成立的背景，大多是出于需求，而不是项目本身的价值。工作的目标过于功利，往往急于去解决某个具体问题，缺乏解决共性问题的意识。

忙，成为一切拖延症的最佳理由。团队在事务优先排序上选错了度量工具，不用价值来衡量，却非要去和时间赛跑。不论是传统运维，还是互联网运维，错误的驱动原力都会让你的团队陷入一个明日复明日的怪圈。

## 6.2 定海神针 CMDB

CMDB (Configuration Management Database, 配置管理数据库) 用于存储与管理企业 IT 架构中设备的各种配置信息，它与所有服务支持和服务交付流程都紧密相连，支持这些





流程的运转、发挥配置信息的价值，同时依赖于相关流程来保证数据的准确性。

在实际的项目中，CMDB 常常被认为是构建其他 ITIL 流程的基础而优先考虑，ITIL 项目的成败与是否成功建立 CMDB 有非常大的关系。

70%~80% 的 IT 相关问题与环境的变更有着直接的关系。实施变更管理的难点和重点并不是工具，而是流程。即通过一个自动化的、可重复的流程管理变更，使得当变更发生的时候，有一个标准化的流程去执行，能够预测到这个变更对整个系统管理产生的影响，并对这些影响进行评估和控制。而变更管理流程自动化的实现关键就是 CMDB。

## 6.2.1 CMDB 是一切运维的基石

如果要问运维团队里最重要的系统是什么，我想毫无疑问就是 CMDB 了。CMDB 一直被认为是 ITIL 服务管理的核心，位于最底层的支持系统位置上，由此可见其重要程度。

### 1. 用 CMDB 去衡量运维团队的能力

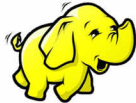
衡量一个运维团队的能力如何，我有一个非常简单粗暴但又行之有效的判断依据，那就是去检阅他们的 CMDB 到底能做到什么程度。

运维团队有时会接待一些前来参观视察的领导团。通常都是在一番欢迎仪式之后，由负责人带领大家来到监控大屏下面，给众人展示那些交易量、成功率、运行状态等各种图表。曲线图平滑优雅，状态面板是一片绿色的海洋。总而言之，我们的交易量很高，系统运行很稳定。众位领导在绚烂的 UI 和美丽的交易数据面前频频点头，时不时还相互耳语几句。随同而来的小秘书，一脸花痴状地望着英俊帅气的运维总监两眼出神。演讲完毕，众人纷纷都给出了满分。

然而，从技术角度上看，这些其实都是虚的。领导的视察时间都是提前选好的“良辰吉日”，参观的时段都是系统运行最正常的时刻。这种表演纯属面向业务，为了提升企业形象，增强投资者的信心而已。如果一个技术团队来造访，还采用这种展示方式，就显得太没有诚意了。

我以紧急故障处理为例。夜半时分，监控团队正在办公室值班。忽然间警铃大作，报告有一台 IP 地址为 192.168.0.250 的应用服务器的网络宕掉了，其他情况暂时不明。监控同学见此情形，迅速拨打了 SE 的值班电话。

值班的 SE 接到电话后得知 192.168.0.250 远程无法登录，但监控没有提供带外管理地址。登录查询系统，输入用户名密码后，发现这个地址找不到。没办法，SE 又拿出自己做的 Excel 表格查询，终于找到了带外管理地址。登录后，经 SE 检查发现，是因为内核崩掉后导致的重启，重启后又由于硬件故障卡在了自检那里。与此同时，监控同学将故障也告知了负责应用运维的 PE 团队。不巧的是，当天 PE 团队的值班人员并不是负责这台服务器的 PE。他和 SE 一样，也是费了一番工夫才联系上了这台服务器的拥有者（Owner）。仅仅



是登录服务器和联系干系人，就花费了两个人十多分钟的时间。

监控平台的告警信息不完整。除了故障现象和业务地址以外，其他的什么都没有，还是要其他人来完成二次查询和确认的工作。那么这个责任在谁？是监控团队的问题么？我认为归根到底还是 CMDB 的问题。既然业务地址已经无法登录，为什么不提供带外管理地址呢？带外地址为什么在查询系统里面无法找到，还需要手动查询 Excel？既然知道故障主机属于某个业务，为什么没有提供业务的负责人呢？监控团队的工作是什么？在故障出现时，监控团队的首要任务是通知和组织干系人。这两项工作不应该再交给其他人去做了。由于监控提供的信息不完整，监控团队的作用根本就没有发挥出来，导致监控的工作是无效的。

这就是我为什么要说，评价一个运维团队的水平要看他们 CMDB 模型的能力成熟度。CMDB 是一切运维的基石。一个楼盖的好不好，首要是地基得牢靠。一辆汽车性能强不强，是由发动机、变速箱和底盘来决定的。光看表面文章没有用。根基差，上层做得再绚烂、再美丽，亦如同浮沙筑屋一般。

对于监控告警平台的要求，不是说故障出现时，能够及时发出警报这么简单，你要把所有相关的信息都发送出来才行。什么信息都没有，光喊狼来了，那是不行的。告警的意义是什么？警报发出来，不是让大家去避难的。运维团队是前线作战部队，警报发出来，就意味着战斗。既然是战斗，那就必须得弄清楚敌人的全部情况才行。

你说狼来了，让我们做好准备。可是我们都不知道狼从哪里来，有几只狼，离我们有多远，你要我如何准备？

## 2. CMDB 的多米诺效应

最糟糕的运维就是在没有打好基础的时候，反倒是先把上层建筑都统统搞起来了，唯独丢掉了最重要的 CMDB，用 Excel 表格来自欺欺人，后期信息缺失全靠人肉支撑。想一想，这是多么可怕的一件事！

CMDB 是运维组件多米诺骨牌当中的第一张骨牌（见图 6-1）。想想看，部署系统、配置管理、监控平台哪一个离得开 CMDB？如果 CMDB 倒下了，那么后面所有的内容都将不复存在。

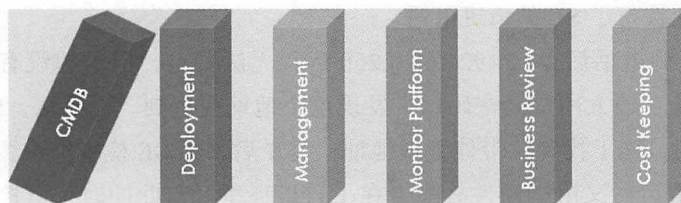
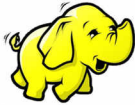


图 6-1 CMDB 是一切运维的基石

然而，先盖楼、后打地基正是某些运维团队现状。这样做的理由就是因为打地基的





成本高，进度慢，所以就先拿一个东西来凑合，剩下的以后再说。反正，这种团队总是有一大堆理直气壮的理由去拒绝一个 CMDB 系统的建设。面对这样的运维态度，我们也只能呵呵了。然而，这口黑锅肯定是要有人来背的，欠下的账迟早要还，只是可怜了那些无辜的背锅人罢了。

### 6.2.2 是什么毁了 CMDB

大多数运维团队的问题在于根基没有打好，一系列错误的观念与行为有可能导致你使用了一个假的 CMDB 系统。

#### 1. 偷换概念的迭代盾牌

一个项目的开始，往往就是类似这样的开场白。我们做这个项目是因为有一个某某问题的出现，为了解决这个问题，我们决定如何如何。一开始不用那么完美，先解决眼前的饥渴，剩下的东西我们可以迭代，有什么问题以后再慢慢调整优化。事实上，这种做法并不是真正的迭代，根本就是摸着石头过河。

#### 2. 闭门造车

软件开发的过程中最忌讳的就是想当然。在构建 CMDB 的过程中，实际上是一个业务逻辑梳理的全过程。因此，CMDB 的整个架构设计应当由业务团队去构建，而不是甩给开发团队。不懂业务就没有发言权。缺少对业务的深度理解，就无法全面准确地定义配置项以及建立它们之间的关系模型。

当然，在定义配置项之前，也需要有切实可行的流程规范渗透在里面，不了解流程的前提下，去做配置项就是空谈。

#### 3. 空口无凭

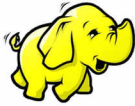
在软件项目的开发过程中，有一个非常大的问题，就是各个团队之间没有文档输出，完全靠 PM 的会议纪要去支撑。

这里面存在两种情况。

第一，缺乏需求文档。需求来源于使用方，然而使用方没有输出需求文档，变成了会议纪要和电子邮件，或者是开发自己在汇总。最后，整个系统设计完成后全部走形。

第二，开发文档不开放。开发团队对写代码这种事情很专业，但是他们对业务逻辑的理解并不那么到位。写程序的过程等同于工作交接，把人所做的工作方法“教授”给程序。程序逻辑必须和人的逻辑一致，所以很多逻辑实现是需要和业务需求方去核对的。比如提取一个信息，作为业务方会更清楚怎么做才是正确的姿势。所以，他有权力向开发了解这个功能是怎么实现的，他的意见也是最权威的。对此，开发团队却认为这样的做法“侵犯”了他们的“领空”。他们认为，我只要实现你的需求就好了，代码怎么操作是我的事情，你管不着。盲目自大导致了糟糕的错误逻辑，在产品上线后，闹出了很多笑话。业务方用着





不爽，要么忍着，要么冷嘲热讽，就此还产生了一些团队之间的不愉快。

### 6.2.3 如何定义你的需求

这是一个没有数据就寸步难行的信息时代。所谓无数据不服务，你很难想象一个没有数据的系统是怎么建立起来。但这又是一个信息爆炸的时代，我们一方面渴望拥有丰富的数据资源，而另一方面却又因为受到各种虚假、冗余的信息轰炸而感到身心俱疲。面对信息的获取，我们缺乏有效的筛选和控制。就像一个拾荒者，疯狂地汲取了大量的垃圾——虚假、冗余和无用的信息，反而加大了信息利用的困难程度。

我们构建一个 CMDB，需要它做什么？我们的目的是存储有价值的信息。关于价值如何去定义，我想可以根据如下这五点去衡量。

#### 1. 关联性

提供的信息一定要和用户有关联，是用户真正希望看到的内容。

我举一个 CPU 硬件信息采集的例子。我们希望在带外管理中，能够看到 CPU 的厂商、主频、产品型号、核心数量等（例如 Intel E5 2650 v2 2.6GHz 8Core）。至于它的内部代号就属于无用信息，因为大多数用户对此并不感兴趣。

再比如说，当一台主机故障无法登录的时候，系统工程师希望看到的是带外管理地址，告警系统只给出一个业务地址是没有任何参考意义的。因为业务地址此时根本没法登录。

#### 2. 准确性

虚假数据比没有数据危害更大，它会导致决策者出现错误判断。虚假的信息往往来源于人工输入和逻辑错误。人难免犯错，如果信息源大量来自于人工输入，势必会生成更多的错误内容。同样，程序在录入的时候也会出现错误，但这种情况多来自于程序设计之初的逻辑错误。具体原因有很多：不了解业务，做了错误的逻辑；考虑欠妥，没有规避特殊情况；技术水平有限，没有使用正确的方法，等等。

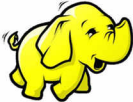
#### 3. 时效性

失效的数据就像一张已经过期的购物卡，即使它的面值是 100 万元，也只能被丢进垃圾桶。

像开头所讲的故事一样，茜茜的数据在一开始是有效的。但由于后来部分信息没有更新，导致过期信息失去了存在的价值。我们在设计之初就应当约束——所有的数据变更都要流经 CMDB，不能出现故事中所讲的那种情况。这就是时效性的要求。在某种程度上说，失效过期的数据也是一种虚假的信息。

#### 4. 精练性

一条信息如果不断地重复就变成了广告，所以去重也是很重要的。最典型的例子就是告警信息。没有告警信息显然是不行的，但是告警信息太多会让人失去耐心。正所谓“虱



子多了不咬、债多了不愁”，过量的告警不会令人更加重视问题，反而倒逼别人想尽一切办法去屏蔽信息来源。

信息去重的方式有两种。第一是汇总。例如，日志里有 100 次关于 too many attempts 的尝试性登录记录，正确的告警方式是发布一条关于 too many attempts 的通知，并汇总这个信息出现了 100 次。而不是连续发 100 条通知。第二是升华，进行分级处理。通告系统有恶意入侵的行为，具体的详细情形采用二级信息的方式展开：告知用户我们发现 100 条尝试性的错误登录，有 1024 个端口扫描，有溢出攻击行为，等等。

关于在 CMDB 上面去重的具体应用，我们以硬件信息为例。一般硬件信息清单的获取都是根据系统总线扫描得到的，像内存条通常会列举出每一条 DIMM 信息。在存入 CMDB 之前，可以精简为单根容量、数量和总计容量这三个内容，而不是把底层信息原封不动地抄送进来。即便如此，在数据展示的时候也没有必要把信息全都平铺出来。人脑对信息数量的记忆是有限的，给的太多，反而让人挑花了眼，什么也记不住。

### 5. 独特性

信息提供切忌出现常识性内容——给的都是你不要的，说的都是你知道的。

相声泰斗马三立先生有一个著名的单口小段叫《祖传秘方》。讲的是一个人在大街上卖祖传的止痒秘方，而且是立即见效，不灵不要钱。有一个患者，花了五毛钱买了一份，回到家里打开一瞧，结果里面只有两个字——挠挠。身上痒了用手挠，这是人之天性，莫说是秘方，恐怕连常识性的知识都算不上。这个相声小段在抖开包袱之前做了很多的铺垫，最终这样一个结局令观众忍俊不禁。

艺术作品让人莞尔一笑，然而现实中这种结局多是令人气愤和失落的。比如，你在执行一个操作时出现了异常，想通过查询日志进行分析时，却只在日志中看到了这样一条——“操作出错，请你联系管理员”。想来这种日志不要也罢。我们在测试服务器硬件清单抓取的工作中发现，一些服务器产品提供的存储信息非常简陋，竟然只是告之哪些插槽上有硬盘，连容量和 SN 都看不到。这种就属于完全没有价值的信息。

## 6.2.4 如何定义表结构

### 1. CMDB over IaaS 的信息定义

我们可以根据信息的分类，把 CMDB over IaaS 的信息分成三大类：第一类是资产信息，第二类是业务信息，第三类是通用信息。前两种顾名思义，自不必再论。对于通用信息的定义，这里面包括了地址信息和设备使用的状态。因为从各种角度讲，它们横跨了前两种信息的领域，地址既是业务信息，又可以认为是一种资产信息。当设备没有被使用时，其状态更接近于资产；如果上线了，又比较靠近业务一端，有点儿“脚踏两只船”的感觉。因此，我便把它们归类到通用信息。表 6-1 所示为信息定义表。

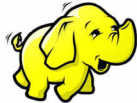
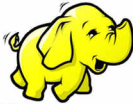


表 6-1 信息定义表

| 信息字段             | 信息描述           | 收集方法            |
|------------------|----------------|-----------------|
| 资产信息             |                |                 |
| Product_SN       | 服务器 SN 号       | IPMI            |
| Assets_SN        | 内部固定资产编码       | IPMI            |
| Vendor           | 服务器生产商         | IPMI            |
| Type             | 服务器型号          | IPMI            |
| Classes          | 内部服务器类型定义      | IPMI            |
| HW_Inventory     | 硬件配置清单（包含多个字段） | 带外管理或系统采集       |
| IDC              | 数据中心名称         | LLDP            |
| Address          | 数据中心地址         | 人工录入            |
| Consignee        | 收货人（存在多个值）     | 人工录入            |
| Mobile           | 收货人手机（存在多个值）   | 人工录入            |
| Email            | 收货人邮件（存在多个值）   | 人工录入            |
| Location         | 机房房间号          | LLDP（自定义字段）     |
| Rack_No          | 机柜号码           | LLDP（自定义字段）     |
| Unit_No          | 机架位置           | LLDP（端口号对应上架位置） |
| Warehousing_Date | 入库日期           | 自动采集或人工录入       |
| Lifecycle        | 生命周期           | 一般为三年           |
| Retirement_Date  | 退役日期           | 自动采集            |
| Price            | 整机价格           | 人工录入            |
| 通用信息             |                |                 |
| Business_IP      | 业务段地址（包括宿主机地址） | IaaS/OS         |
| Outofband_IP     | 带外管理地址         | IPMI            |
| Device_Status    | 设备状态           | Workflow        |
| 业务信息             |                |                 |
| Res_Type         | 用于区分物理服务器和虚拟机  | IaaS 或系统采集      |
| VLAN_ID          | VLAN 号         | IaaS 或系统采集      |
| Business_NAME    | 业务名称           | Workflow        |
| FT_Rank          | 定义故障容错等级       | Workflow        |
| Owner            | 申请人            | Workflow        |
| OwnerDept        | 申请人所属部门        | Workflow        |
| Checkin_Date     | 申请日期           | Workflow        |
| Checkout_Date    | 退还日期（默认退役日期）   | Workflow 或预定义   |
| Using_Period     | 使用周期           | 自动采集            |
| Cost             | 成本核算           | 自动采集            |





收货人主要是指数据中心的驻场工程师。一般设备上架、硬件派单维修等工作，都需要数据中心驻场人的联系方式，因此收货人及其联系方式应当收录到 CMDB 当中。硬件报修系统在发送报修申请时，可以结合 CMDB 把干系人的联系方式连同故障邮件发送出去，便于厂商派单，而不需要再由报修人手动填写这些信息。信息都应当包含多个数值，因为联系人有可能不止一个。

LLDP (Link Layer Discovery Protocol, 链路层发现协议) 是一个非常好的二层协议。它允许网络设备在本地子网中通告自己的设备标识。LLDP 本来是为了让不同品牌的网络设备在网络中相互发现及信息交互使用的。它可以将设备的管理地址、接口标识等信息组织成不同的 TLV (Type/Length/Value, 类型/长度/值), 并封装在 LLDPDU (Link Layer Discovery Protocol Data Unit, 链路层发现协议数据单元) 中发布给与自己直连的邻居, 邻居收到这些信息后将其以标准 MIB (Management Information Base, 管理信息库) 的形式保存起来, 以供网络管理系统查询及判断链路的通信状况。

我们可以借助这个手段, 在部署服务器之前, 将一些我们需要的基础信息通过它传递出来。比如带机柜号的设备标识、服务器对端的端口号标识等。至于机架的位置, 没有必要去确定 U 位。我们可以通过定义端口与服务器上架顺序之间的映射关系来实现 U 位的定义。比如, 按照从上到下的顺序, 第一台服务器接入交换机第一个端口, 这样通过端口来确认摆放次序, 加上 SN 和资产编号的核对, 可以很快地定位目标主机。

除了资产信息以外, 业务信息大多由 Workflow 系统的填报录入。因为业务信息不像资产信息那样, 大部分可以依靠技术手段去获取。要尽早地丢弃 Excel 记录的方式。人工记录信息往往是离散的, 资产信息可以由 SE 协助完成, 而业务信息是依靠 PE 去操作的, 无法形成统一的信息库。而且随着工作交接, 表格越传越走形。往往到最后发现信息不准确, 再到处去找人核对。有些离职了, 还不一定能够找到当事人。

而 Workflow 的数据是一开始发起业务请求时就准备好的, 业务请求不通过 Workflow 就无法执行, 这样可以确保所有的系统数据都记录在案, 最后才能实现有据可查。

FT\_Rank 指的是容错等级, 它是一个百分比数值, 代表了你对故障等级的容忍程度。当然, 如果你愿意, 也可以增加一个 B\_Rank (业务等级), 用来标识业务的重要性。不过, 我个人认为这个标识不太适合在 IaaS 级别的 CMDB 中出现。对于一个运维团队来讲, 任何业务都是重要的。但是, FT\_Rank 则不一样, 它在故障出现时反映了事务的紧急程度。这需要在业务部门提报服务器申请的时候一并提交, 表明该项业务在运行期间如果发生故障, 允许损失百分之多少的访问量。

假设你申请了 10 台服务器去构建一个 Web 服务, 并允许 10% 的访问量下跌, 那么 FT\_Rank 就是 90。如果在运行期间有一台服务器宕机, 就应当属于紧急事件了。此时已经达到了当初的预设值, 不能再出现第二台故障机。

相反, 你只允许访问量下跌 5%, 那么 FT\_Rank 就是 95。但是你目前有 100 台服务器,

后者的容错等级虽然高，但是发生故障时要比前者显得更加从容一些。因为你可以接受最多 5 台服务器同时宕机。所以，FT\_Rank 和你的业务重要程度无关，我们只关注在某一个时间点上服务节点的可用率，而不管业务本身是否重要。当同时发生多个业务故障时，我们可以根据实际情况去调整处理的顺序。

当然，这个算法需要你自己去评估和定义。同样，监控平台的告警系统也可以根据节点可用率与 FT\_Rank 的差值去定义告警级别。在这里，我们把 FT\_Rank 定义成节点可用率的百分比，允许 10% 的访问量下跌，那么 FT\_Rank 的值就等于 90，我们通过监控的 Agent（代理）可以实时了解有多少台存活的服务器，同时根据 Business\_Name 可以统计这个业务总共有多少台服务器。这两个数据相除之后所得到的结果：如果低于 FT\_Rank 就应当发送一个 Emergency 通告，属于必须马上处理的问题；如果远高于 FT\_Rank，则可以作为 Critical 或者 Warning 来发送。

为什么要进行成本核算？这是一个出彩的地方。运维团队负责提交采购申请，但设备大多不是给自己使用的，而运维团队对各部门的实际使用情况根本不清楚。每次采购提报的时候，都会被财务部门质疑为什么要花费这么多钱。如果能够拿出各部门的使用情况报表，可以给决策者提供一些有价值的参考消息。老板一看此表，就知道今年哪个部门花了多少钱。如果这个部门无法产出效益，还申请了很多的设备，来年采购就会被限制。如此一来，就无须再让运维团队去背锅了。

成本计算公式：

$$\text{成本} = \text{单机价格} \div \text{生命周期} \times N \times \text{使用周期}$$

服务器生命周期一般为三年。计算物理机时， $N=1$ ；计算虚拟机时， $N=$  租户使用虚拟机的个数和该宿主机中虚拟机总数的比值。

当然这里所涉及的 CMDB 信息远不止于此，我们只谈到了与 IaaS 相关的一部分内容。在后面我们会讲，CMDB 应当是分层设计的。关于面向应用的信息不在本次的讨论范围之内，因此就不再过多论述了。

## 2. 设备状态表

服务器的设备状态是一项非常关键的信息。服务器从供货上架到最终下线报废的整个生命周期当中，状态也在不断地发生着变化。在执行一次业务操作之前，不论是采购、扩容，还是日常操作，目前已用设备是多少，空闲资源池的空间如何，对于这些情况的掌握是十分必要的。表 6-2 所示为设备状态表。

## 3. 硬件清单信息表

硬件清单信息列表可以单独放置。由于大部分场景很少会用到该表中的全部信息，因此该表无须和其他内容混放在一起。更多的情况下，该表是独立使用的，或者用于定义 Classes 内部服务器类型。对于系统自动化部署来说，我们完全可以使用 Classes 来定义不

同机型，部署不同的系统模板。由于硬件清单涉及多个组件，有些属性是通用的（如品牌、型号和数量等），而有些属性是独立的（如 CPU 的主频、网卡的 MAC 地址等），这就将涉及一个多维度的处理问题。最好是每一个组件设立一张表，然后和 Product\_SN、Assets\_SN 实现关联。

表 6-2 设备状态表

| 状 态        | Owner  | 描 述                |
|------------|--------|--------------------|
| Raw        | SE/IDC | 服务器已加电，但未部署操作系统    |
| Ready      | Null   | 操作系统已部署完毕，等待分配     |
| Assigned   | All    | 服务器已分配给 Owner，但未上线 |
| Online     | All    | 正在运行的线上系统          |
| Recyclable | SE/IDC | 系统已下线，可以回收的服务器     |

#### 4. 数据中心联系人表

数据中心联系人表（见表 6-3）是日常数据中心工作要使用到的列表。这部分信息完全是需要手工填写的。用 IDC 作为数据中心联系人表的主键，在机房负责人和服务器之间建立一套映射关系。这套映射关系对于发起硬件报修来说至关重要。

表 6-3 IDC Consignee

| 信 息       | 类 型                         | 描 述    |
|-----------|-----------------------------|--------|
| IDC       | Primary key, varchar ( 16 ) | 数据中心名称 |
| Address   | varchar ( 64 )              | 数据中心地址 |
| Consignee | varchar ( 8 )               | 收货人    |
| Mobile    | varchar ( 16 )              | 收货人手机  |
| Email     | varchar ( 64 )              | 收货人邮件  |

### 6.2.5 设计思想原则

幸福的家庭都是相同的，不幸的家庭各有各的不幸。

——托尔斯泰《安娜·卡列尼娜》

好的产品，不管外在的形式如何变化，设计理念往往都是惊人的相似。这就是我们常说的“英雄所见略同”。

#### 1. 预留无罪

关于 CMDB 构建上，有很多反对“高大全”的声音，而笔者却不这么认为，CMDB 就是要“高大全”。我们提倡化繁从简，但需要减掉的是枝蔓而不是主干。



关于 CMDB 的重要性我们已经讲得很清楚了，没有必要在这里重复强调。什么叫作基石？就像习武，内力是根基，内力不深厚，练什么武功都是白搭。内力是怎么来的？不是靠看武功秘籍或者喝毒酒蛇血，而是依靠不断地积累所得。CMDB 要积累的就是数据，要想积累足够有价值的数据，在表结构的设计之初，就要尽可能地预留出足够的字段用于存储有价值的信息。字段越丰富，表达能力就越强，未来所能实现的功能也就越多。例如，上述的成本核算就是一个很好的例子。

孟尝君门客三千，既有贤士冯谖，亦有鸡鸣狗盗之辈。然而，当他困于秦国之时，恰恰是这两个为众人所不齿的鸡鸣狗盗之徒，救了主人的性命。很多时候，我们自认为没用的东西，并不是真的没有用，而是没到用的时候。我们说 CMDB 存储的是有价值的数数据，这些数据在短期内也许可以不用，但是不能够没有。充分的预留是没有过错的。数据用时方恨少，别做那种“平日不烧香，临时抱佛脚”的事情。

## 2. 分层有理

在 CMDB 的建设中存在很多争论。有些观点认为 CMDB 是纯业务的应用，也有些人把 CMDB 和资产管理混为一谈。然而，CMDB 并不是一个平面的架构，CMDB 也是要分层级的。

CMDB 不是一个平铺式的设计，不能错误地把 CMDB 当成一个大开间的厂房来使用。从云架构的角度上讲，我们把云自顶至下分成了 SaaS、PaaS 和 IaaS 三个层级，每层所应对的场景、用户群和需求都不同，CMDB 在每层上需要存储的数据也不一样。要像划分功能区一样，将 CMDB 进行水平切割，不同层级的 CMDB 应由不同的团队来构建。不要妄图把所有东西全都堆砌在一个层面里。到头来，发现这个骨头太大，啃不动，于是乎各种拖延的借口也就纷至沓来。

本书是以运维为主题，围绕着系统这一层来讨论的，因此我们所谈论的 CMDB 应当属于 IaaS 这一层次，这里主要关注的是基础平台构建的部分。对于不同层级的 CMDB，应当交付给不同的团队进行设计。

## 3. 无关大小

不要觉得“CMDB 是只有大企业才需要的，小公司就无所谓了”。

现存的哺乳动物都有七块颈椎骨，不论是长颈鹿，还是小老鼠，它们都是一样的，这是进化最佳形态的结果。有句话讲：“麻雀虽小，五脏俱全。”同样，CMDB 作为基础系统，它的存在与否和你的企业规模毫无关系。

## 4. 扫好自家门前雪

CMDB 不会主动去做什么事情，它只负责被动地接收存储数据，所有的数据修改是由业务变更而触发的。不要反其道而行之，试图通过变更 CMDB 去发起一项业务。

## 5. 数据会腐败变质

数据是有时效性的，数据产生后不能等，要第一时间放进 CMDB。

但是,现实情况存在一些难以回避的问题。就是当你的 CMDB 和 Workflow 还没有构建好,或者周边系统与 CMDB 尚未形成对接时,业务已经开始迫不及待地要开始运转了,使得很多重要的数据无法传递进来。这时候该如何是好?这就是我们为什么在很长的一段时间里始终在用 Excel 负隅顽抗的根本原因。

你要考虑这个时间段的信息怎么留存下来,是你第一时间应当解决的问题,甚至比如何构建 CMDB 和 Workflow 更为重要。Email 是最差劲的数据保留方式,哪怕是使用纸质单据,也不要再用 Email。不要嘲笑这种做法,在无纸化办公之前我们都是这么做的,包括我们现在的人事档案依旧如此。你该嘲笑不是替代方案有多低级,而是你为什么没有及时把 CMDB 和 Workflow 等系统对接起来。

这段时间之内,尽可能使用可靠的替代方案和可靠的专人去管理数据,并速战速决,解决问题,尽快结束尴尬期。

## 6. 不许偷吃

CMDB 应当和任何有可能影响数据变化的系统存在对接接口,确保每一项业务变更都可以通知到 CMDB。数据的增、删、改、查都必须要通过 CMDB,不允许从外部变更和获取数据。我们以业务扩容来举例说明。CMDB 需要和 IaaS 系统进行对接,部署工作由 IaaS 完成并将采集到的结果数据向 CMDB 传递,不应出现手工安装或人为改写数据库的行为。

## 7. 迭代不是挡箭牌

在软件项目开发过程中,迭代往往成为偷懒和逃避的挡箭牌。经常听到关于迭代的一种声音就是——“某某模块开发成本高,影响项目进度,我们放到二期去做”。我们要搞清楚迭代应该迭代什么,以及哪些内容是不能迭代的。

功能可以迭代,但是数据是不能迭代的。一些功能上的开发,你可以后续再慢慢增加,但是数据到后期是很难再去补救的。功能都是基于数据来的,如果数据是缺失或者失准的,那么功能也就无法使用。

架构可以迭代,但架构改动涉及系统整体,所以它的迭代工作要比功能迭代复杂得多,也不可能是随心所欲的。就像我们前面谈论到的表结构的设计,如果要在后期对表结构进行修改,则会带来很多麻烦。

迭代的前提是,要想清楚以后会发生什么问题,并且已经做好了应对的准备,具有平滑过渡的能力。而不是摸着石头过河,以各种理由(项目进度、人员工时、架构复杂)作为迭代的借口,走一步看一步,出现问题后反复修改,增加各种各样的东西,重构场景不断。其最后的结果就是数据全部断层:早先数据已过期,中间数据大概其,最新数据未上线,无据可查干着急。

请不要拿“迭代”作为不可用产品的挡箭牌。

## 8. 杜绝两只手表

一项信息的数据来源永远有且只能有一个。我们应当尊重系统录入，尽可能地采取这种录入方式。人工录入的错误难以完全消除。比如，服务器的 SN 号可以通过 IPMI 协议去抓取，那么 SN 这项信息就走 IPMI 这条采集通道，不要再增加一个允许导入 cvs 的功能存在。多路径信息源无益于校准信息的准确性，相反令人更加不知所措。当出现不一致的情况时，没有办法证明哪一个方案是对的。如果能证明 A 方案得到的结果比 B 方案更准确，那么 B 方案又为何要存在呢？

一只手表可以告诉你当前的时间，然而当你同时拥有两只手表的时候，无从判断到底哪一只表是准确的。同样，对于信息源的来源，你要做的就是选择其中一只表，最大程度上去校准它，并以此作为标准，听从它的指引行事。系统录入要做好采集逻辑，确保使用正确的方式获取数据。遇到必须采用人工录入的信息，应当使用工具或者流程规范去约束录入行为，从而降低出错的可能性。

## 6.3 多面娇娃 Workflow

Workflow 的概念起源于生产组织和办公自动化领域。它是针对日常工作中具有固定程序的活动而提出的一个概念。目的是通过将一个具体的工作分解成多个任务和角色，通过一定的规则和过程，约束这些任务的执行和监控，以达到提高企业生产经营管理水平的目的。

简单地说，Workflow 就是一个根据标准规范而实现的（半）自动化管理过程。

Workflow 和 OA/ERP 系统不同，固化的业务流程非常不利于业务流程的改变。企业要不断地改进自己的管理，实施流程再造，需要一个功能可重构、流程可改变、高度柔性的系统。为此，引入 Workflow 就成为必然的结果。

### 6.3.1 一份周报中竟然 80% 的工作量都是在沟通

---

一份来自野比大雄的工作周报。

1. 完成 Python 生产和开发环境配置；
  2. 针对 Python 运行环境隔离给大胖答疑；
  3. 网络丢包问题排查和协调；
  4. 和杉木就有关硬件和系统信息接口的沟通达成一致；
  5. 和强夫就域名 xxx.com 的应用场景和申请方法进行沟通；
  6. 和静香就域名 yyy.cn、zzz.com.cn 的申请方法进行沟通；
  7. 其他指导、答疑。
- 

通过大雄的这份工作周报，我们可以看到其中存在很多问题。大雄在完成了 Python 的



环境配置之后，没有发布使用标准，而是只给大胖做了答疑，那么静香和强夫如果也有疑问该怎么办呢？还有，他和杉木就标准达成一致之后，竟然没有产生任何文档，那么如何能确保这个标准可以被所有人一直认真地执行下去呢？关于 DNS 域名申请的工作也是同样的问题。

大雄的工作看似很忙，也很努力，他在不停地和每一个人去做沟通，但实际上这些全是重复性的、针对个体的无效劳动。这样一个一个地沟通下来，什么时候是个头呢？究其原因，是他没有产出标准化文档，没有通过推广规范来消除无效沟通，同时缺乏有效的约束工具，来降低答疑的可能性。

### 6.3.2 Workflow 能干什么

Workflow 是一个遵守规则并融于流程之中的（半）自动化系统，它可以确保在整个业务活动的过程中，我们的所有操作都是规范的。角色、设备、事件、状态等是业务活动过程中必不可少的几个关键因素。通过事件来驱动主谓宾的关系，谁做什么，结果怎样。

使用 Workflow 前，首先要明确几个问题。它们分别是：What——干什么？Who——谁参与？How——怎么干？Result——啥结果？

#### 1. What

在一项业务活动中，需要定义完整的业务流程。如何开始？什么时候结束？中间都需要做些什么？

#### 2. Who

任何一项业务活动中都离不开人，不同的人（角色）在业务活动中需要承担不同的责任。大多数情况下，Workflow 把角色作为节点，而流程就是游走于各个节点之间。

#### 3. How

在任何业务活动中都需要设定规则和约束条件，它们决定了流程的最终走向。根据规则和约束条件，可以描绘出某一个业务流程的行走路线图。

#### 4. Result

流程不断往返于角色之间。当一个事件从角色 A 转移到角色 B 的时候，意味着该事件的一个阶段的结束。此时，Workflow 需要告知干系人。在整个流程结束后，Workflow 还要负责发起通告和返回数据结果。因此，Workflow 需要和相关的系统完成接口调用或数据交互。Workflow 要提供应用接口规范、标准的 API 函数，确保在与不同的应用系统进行交互时，可以建立灵活的调用渠道。

### 6.3.3 Workflow 是实例化的规范

讲一个关于规范的笑话：大多数的企业里面，至少有一半以上的工作是没有规范的。

有规范的有一半人都不看规范，看规范的有一半人不遵守，想遵守规范的有一半人不会做，会做的有一半没效果，有效果的没有受到领导表扬，而受到领导表扬的全部遭到群众的坚决反对。

对于任何一件工作来说，规范都是很重要的。它是用来保障标准执行的。但是规范往往又是被很多人所厌烦的东西。你费尽心力写好的规范，往往是没有人去看的。遇到问题以后，大家还是习惯问来问去，或者干脆按照自己的喜好来执行。原因就是根本看不懂，也没那个耐心去看。看规范是一种美德，但是这个美德无法绑架绝大多数人。直至今天，还是有很多人抱怨大公司流程多，工作效率低。这就是规范执行彻底失败的表现。

好了，这个时候该 Workflow 出场了。作为一个遵守规则并融于流程之中的（半）自动化系统，它本身就是标准和规范的化身。我们跟随 Workflow 行事，就等于遵守了规范。

#### 6.3.4 Workflow 是领航员

既然 Workflow 是实例化的规范，那么它就该严格遵守规范才对。我们可以认为 Workflow 是一个导航员，它应当能够引导用户轻松地完成所有的工作。

然而，很多 Workflow 在设计的时候缺乏应有的严谨态度。在用户执行操作 Workflow 时，往往产生很多疑问。用户不知道某些选项该怎么填写。要么去问别人，要么自己随心所欲。前面所提到的故事再一次重演了。

All input is invalid.

——Bill Gates

Workflow 作为一个实例化的规范，应当永远避免 InputField 的出现，除非迫不得已。InputField 需要用户参与输入，而输入是没有约束条件的，这意味着用户可以随便填写。如果这个填写的内容本身是需要约束的，那么这个 Workflow 就是一个彻底失败的作品。假如我们构建了一个 Workflow，用来支持日常工作中的各项业务申请。这时候，有一个用户想要申请一个 DNS 域名。Workflow 的第一步，是要求用户提交申请项目类型的名称。那么就应该给出一个关于各项业务请求标准命名的 List 让用户去选择。如果使用 InputField 让用户去填写，就会出现各式各样、五花八门的名字。例如：

- ☐ DNS 工单申请；
- ☐ 域名解析申请；
- ☐ 名称解析申请；
- ☐ DNS 申请；
- ☐ 上网申请。

最后一条并不是用来搞笑的。如果 DNS 无法解析，确实是不能上网的。你不能指望所有的用户都是行家，所有的用户都说行话。规范是给专业人员使用的，Workflow 是给终端

用户使用的。由于我们假设用户都不会去看规范，那么 Workflow 必须负责引导、约束用户走正确的路，不要让他们自由发挥。否则，Workflow 就没有起到规范应有的作用。

既然用户的输入是有害的，而我们在执行操作的时候又必须有数据输入，那么如何确保输入的准确性呢？我认为可以采用如下这两条原则。

- 使用约束确保语法的准确性；
- 通过审核确保内容的准确性。

任何一个系统都不能完全地避免错误输入。因此，Workflow 需要确保的是语法的准确性。这里是指描述信息的规范性和准确性。以刚才的申请 DNS 域名为例，Workflow 应当定义好所有的业务名称让用户来选择，而不是让他自己去填写。当你给出一个 InputField 文本输入框时，他就不知道该怎么办了。如果采用 List 下拉菜单，用户是不会连 DNS 和 Firewall 都分不清的。这就是约束的方法。假设你需要统计 DNS 工单的工作量，在业务名称不固定的情况下，这项统计工作就无法实现。

有些 InputField 是不可避免的，比如 DNS 域名。这就需要严格的语法检查，而且尽量减少可以输入的内容。假设公司的域名是 example.com，我想申请一个域名 apps.example.com，那么就让用户只填写 apps 好了，后面自动跟上 example.com，不要让他去填写 apps.example.com。如果有子域，那么子域的选择需要使用 List。你看，你在公网上申请一个域名不就是这么做的吗？容易产生疑问的、不清楚的东西多使用 List，不要在旁边弄一大堆注释试图让用户去阅读。永远不要让用户去琢磨——这个内容我该怎么填？

至于内容的准确性，Workflow 是保障不了的。把申请地址 1.1.1.1 写成 1.1.1.2 这种错误只能通过人工审核来解决。程序不是神，人都判断不了的东西就不要让程序来做主。

### 6.3.5 Workflow 设计中的常见问题

和 CMDB 一样，Workflow 在设计过程中也存在着各种各样的问题。不过平铺直叙到这里，我自己也觉得有些烦躁了。不如这样，我们这次来个特别的安排，一边讲笑话一边分析问题。当然，这是一个根据关于需求变更的经典笑话改编而成的，也许你看过第一个，但我保证你没有看过后面几个。

---

#### 运维故事 11：宫保鸡丁

##### 1. 讨论功能而非流程

小二：有位客官点了一份宫保鸡丁。

厨子：好嘞！

厨子做到一半……



小二：客官说菜里不要放肉。

厨子：我肉都回锅了……

然而，厨子还是一点点把肉挑出来了。

又过了一会儿……

小二：客官问菜里能给加点儿腐竹吗？

厨子：你不知道腐竹得提前泡水？炒到一半才说？你去跟他说，想吃腐竹就多等半天。

小二：啊，你怎么不早说？

厨子：我怎么知道他要往官保鸡丁里放腐竹。

然而，厨子还是去泡腐竹了。

又过了一会儿……

小二：客官问菜里有没有加茄丁？

厨子：加了茄丁，那还能叫官保鸡丁吗？哪个店家这么做菜？

又过了一会儿……

小二：客官说还是加上鸡丁吧。

厨子：你这不是玩我吗？你顺道问问他，腐竹还要不要？不要我就扔了。

客官：咦，怎么这么慢？反正还没付钱。那个什么，小二，菜我不要了，退了吧。

---

Workflow 在设计时，需要考虑实现的是功能而非流程。流程并不是一成不变的，往往需要在使用过程中不断地优化和调整，展开讨论并实例化一个 Workflow 是无意义的，需求的反复变更是令人崩溃。不论怎样，实例化的 Workflow 都没法满足你的业务需求。因为，在你的开发周期还没有结束的这段时间，需求和流程可能已经变更好几次了。

因此，Workflow 管理系统在设计和实施中，需要提供足够的柔性，来满足不同应用的需要。完成组件化的功能模块，让用户自己去构建流程。而不是把所有的流程全部写死。让厨子负责提供组件，把鸡丁、茄丁、腐竹统统都做好。至于怎么搭配，让食客自主选择。

## 2. 无“锁”不能

---

小二：有位客官点了一份官保鸡丁。

厨子甲：好嘞！

厨子乙：好嘞！

过了一会儿……

小二：客官，你要的两份官保鸡丁来啦，请你慢用。

客官：我只要了一份，为啥端了两盘上来？怎么着，想黑某家？说话间，探臂膀伸手拉出了二十八斤重的鱼鳞紫金刀。

---

Workflow 在下发 Case 的时候，往往是面向一个角色、一个群组而非一个具体的自然人。两个厨子相互之间是没有沟通的，所以有可能出现多人操作的情况。

假如这个操作是幂等的（操作多次和操作一次的影响是相同的）则无所谓。反之，多次操作就会出现很严重的问题。

程序开发时，必须要考虑并发操作所带来的影响，这是起码的意识。在 Case 派发的时候，必须有“锁机制”。实现方法可以使用 Case 认领的方式来解决，只有认领的人才能看到任务内容。如果一个 Case 已经被人认领了，那么它将被锁定，不能进行第二次点击，除非 Case 的状态被修改（比如放弃或者转交）。必须注意，Case 的执行在任何一个环节都不能出现多头管理的情况。

### 3. 色即是空

---

客官：小二，给我来一份官保鸡丁。

小二：客官，你这鸡丁是要山鸡、野鸡还是家养的土鸡？

客官：某家自是要家养的土鸡了。

小二：客官，鸡丁是切长块儿，还是方块儿？

客官：啰唆，随你罢了。

小二：客官，你这配菜是要黄瓜丁、花生米，还是胡萝卜丁？

客官：都要都要，你等只管快些上菜。

小二：客官，我们这有李、高、王三位师傅，你要谁伺候你这顿伙食？

客官：滚！

---

把流程设计得太复杂，是一件非常糟糕的事情。切记，少就是多，多即是无。

我们举一个硬件报修流程的例子。我们的团队曾经在讨论由谁发起报修申请的问题上，出现过两种不同的声音。传统观点认为发起人应当是 SE。还有一种观点认为，所有人都可以发起报修申请，这样做可以及时报修。第二种观点就是典型的“画蛇添足”。硬件有没有故障是需要人去确认的。比方说，我把两路电源拔掉一路再插上，对于系统来说就是一次故障，但实际上电源并不需要维修。既然需要人来确认，那么这个角色只能是 SE。只有 SE 是负责服务器管理的，他既是有权限查看的人，也是最具权威判断的人。如果所有人都去发起报修申请，最后还是需要 SE 来确认。采取这样的方式，不但不能及时报修，还凭空多出一个确认环节，岂不是庸人自扰吗？

再举一个关于硬件报修的例子。当 SE 发起报修申请后，需要由 Owner（服务器的业务使用方）进行审批，因为一些故障的维修需要停机，审批的过程也就是告知 Owner 做好准备。只有 Owner 审批完成后，邮件才会发送给厂商。假设这台故障机需要停机维修，Owner 在收到申请的时候有两种选择。第一，Owner 只是审批通过，但什么都不干，等厂商的维修人员到现场后再做停机处理。第二，Owner 先完成业务迁移工作并关停设备，然后审批通过发送报修邮件，厂商到现场后可以直接维修。

第一种做法，厂商的维修人员到了现场不能及时处理，还要等待系统迁移下线。整个确认过程中涉及业务、监控、SE、IDC、硬件厂商五方的沟通，大大增加了角色之间的交互工作。这种流程的发布，会导致各角色之间反复地多次确认。在不管怎样都需要停机的情况下，先做和后做是没有区别的，一个线上系统如果发现硬件问题，应当在第一时间及时迁走，而不是等待。如果业务不能停，是部署规划不到位的问题。如果确实没有条件在第一时间展开迁移工作，那么应当尽快搭建迁移环境，待完成迁移工作后再发送报修邮件。

#### 4. 君臣不分

---

客官：小二，小二，快些出来。

小二：客官，你有何吩咐？

客官：你这鸟店家，某家点了一坛老酒，五斤牛肉，两张大饼，清蒸黄鱼、红烧鹿肉、孜然羊腿、官保鸡丁各一份，现在已过了半个时辰，为何不见上菜？

小二：客官莫急，其他的菜都已做好。只因后院鸡肉不曾备得，掌柜的回家去取了。客官稍待片刻，待那官保鸡丁做好，小的一并给你端上来。

客官：混账，你这厮好生无礼。待到那时，黄花菜都凉了，那还能吃吗？不要走，且先请你吃我一拳。

---

我们还是用硬件报修系统的流程来举例。SE 在发起报修申请的时候，为了合并处理，允许一次提交多个故障设备。Workflow 把所有的设备，作为一个 Case 发布出去了。但是，这些设备不一定一次性修完。因为，有些设备可能没法下线，而有些不是一个供应商，所以如果等待 Case 内的所有设备都修完再终结就会出现逻辑问题。有些设备明明修完了，在流程中却属于未完成的状态。而且 Case 里可能会有很多 Owner，到底谁来关闭这个 Case 也是个问题。

所以 Case 应当是分级管理的。一个 Case 下面包含若干 Task，一个设备对应一个 Task。Case 由报修人创建，Task 随着填写设备自动生成，每一个 Task 完成后由 SE 确认关闭，当所有 Task 都关闭时，Case 自动关闭。



## 5. 死路不通

掌柜：小二，客官点的官保鸡丁为什么迟迟没上来。

小二：鸡肉没有了，所以没做。

掌柜：混账东西，那你为什么不和客官打声招呼，让他换个菜，害得人家在那里等了半个时辰。

小二：不是你说，客户为先，不能 Say No 的吗？菜又做不了，拒绝又不行，你让我怎么办？

Workflow 在没有结束之前，不管流程走到哪一步，都必须能够进退自如，不能丢在一个角落里面就没有下文了。假使一个操作我执行不了，那么应当允许我拒绝执行，让业务状态回退或者结束。一些 Workflow 系统的“官本位”设计理念非常严重。往往是审批这一步有拒绝权限，而审批人面对申请从来都是看也不看，一律“同意、谢谢、请支持”。在执行环节，执行人在面对错误的请求时无法拒绝，只好到线下去找申请人进行沟通。

## 6. 名词乱入

客官：小二，这道“火山喷雪”看名字煞是有趣，快些给某家来上一份。

小二：好嘞！“火山喷雪”一盘，请你慢用。

客官：我去，这是什么？你这厮竟敢戏耍某家，这不是西红柿拌白糖么？

Workflow 作为规范实例，在业务描述中要使用该业务领域的专有名词。当行家要说行话，用词必须要准确，不能去自己造词。比如，我们有两个不同 ISP 的数据中心，要在它们之间构建一套 DNS 系统。为了实现流量划分，根据请求的客户端地址范围分别提供不同的域名解析结果，实现不同线路上的用户划分到不同的数据中心去。这就是 View 的概念，View 就叫 View，而且只能叫 View，你不能叫“分组解析”等其他名字，因为在 DNS 里面没有你这个所谓的分组解析的概念。乱起名字会引起混淆，有可能会引发操作者误操作。

## 6.4 本章小结

本章讨论了关于 CMDB 和 Workflow 在运维平台建设工作中的重要性。

CMDB 是一切运维的基石，存储了所有日常工作中必不可少的、有价值的信息，所有上层系统的完美运行都离不开它的支撑。就某种程度上而言，CMDB 模型的能力成熟度决定了运维团队的水平。运维团队的管理者应当对此给予足够的重视。CMDB 在建设过程中需要分层设计，不同层级的 CMDB 交由不同的团队去构建。在表结构设计的时候，要充分

考虑今后业务使用的需求，尽可能预留出足够的扩展空间，相关数据要确保在第一时间存入 CMDB，不能在后期依靠人工补救。

Workflow 是规范的实例化体现，它是一套嵌入流程的自动化工作系统，主要的作用是确保用户执行过程的标准化、规范化。同时，我们重点讨论分析了在 Workflow 的设计当中存在着的模块化设计、并发机制、名词乱入、无用设计等一系列问题。希望通过这些实际案例的介绍，给读者在设计 Workflow 系统的时候提供一些帮助。

下一章我们将进入基础服务篇，为大家介绍系统部署、DNS、NTP、配置管理和文件共享方案。

## 构建 IaaS 平台系统

我在进入互联网行业之前，其实从来都没有机会接触自动化部署。以前的我对系统部署是深恶痛绝的。这种观念来自对数据中心的恐惧和因职业被人误解而产生的羞耻感。每当我深陷在数据中心的噪声和辐射当中埋头苦干，辛辛苦苦的工作不过是为给别人安装一台 Windows 2003 或者 RHEL 5。一个几十分钟的活儿要耗费大半天的时间去找授权、安装介质和驱动程序。工作完了还被关在带有门禁的楼道里，等着甲方来接我出去。最后再被人冠以一个“装机的”的称号，确是有些悲愤交加的。

那时候，自动化部署技术对我来说一直非常遥远。因为我很少会遇到五台以上的部署需求。与其浪费时间构建一个部署平台，还不如自己手脚麻利点儿来得实在。为此，我也曾经自己封装过 Linux 版本的自动化部署 CD，以此来平复我的焦虑心情。当进入互联网行业之后，我才真正体验到了自动化部署的优势。在一次紧急扩容的任务中，从收货、上架，到网络布线、服务器初始化，再到系统部署，我只用了两天时间就完成了 160 台服务器的交付工作。经过那一段时间的反复磨炼，我总共部署了将近 10 000 台的物理节点，前后处理了大小几十个大小问题。这让我对系统部署也有了一个全新的认识。过去觉得无聊、厌烦的事情现在开始逐渐变得有趣，面对各种挑战，也不时会产生一些新奇的构想。从为了完成高效交付到逐步思考 IaaS 平台构建，我发现简单的小事情也可以通过创造带来艺术的气息。

当时，刚从支付宝入职的运维总监向我问起基础平台交付能力的时候，他给了我一个“一周一万台”的指标。我当时就告诉他说，如果特别紧急，允许我稍微加一会儿班，那就一天一万台吧。后来“一天一万台”成为我们运维部的一个诨，经常被人拿来调侃，比喻工作效率极高。但我觉得“一天一万台”也没什么可值得炫耀的。到目前为止，我们的保有总量还没有突破两万台，根本也不存在这种场景。日交付指标是要让 IaaS 平台具备这样的能力，在未来业务飞速扩张时有足够的底气去应对。作为基础架构平台的建设，要考虑得长远一些，门槛自然就要高一些，因为底层不能总是改动。

本章将为大家介绍如何构建一个 IaaS 平台系统。我们将从业务流程入手，针对解决



方案的选择、服务器初始化设置、后端部署系统的实例化构建、前端各组件之间的调用和 KVM 虚拟机等几个方面，详细分析构建过程中所遇到的种种问题，并提供一些实际的解决方案供读者参考。

---

## 运维故事 12：一气呵成

“啥？这 1000 台服务器，你们打算今天就全部安装完？”

一听这话，老胡眼睛瞪得大大的，脑袋摆得像拨浪鼓。显然，他是不相信小付的话。“我还跟领导说，我们加班加点最少也得搞上一周呢。你们这些年轻人，话可不要说得太满啊。”

“老胡，你就放心吧。要不然，咱们打个赌，下班之前我俩要是搞不定，就请你喝上一个礼拜的酒。”

“好，一言为定。要是我输了，这周的饭局我全包了。”

眼瞅着下班的时间到了，只见小张和小付俩人还没什么动静。老胡笑了，年轻人就是嘴上没毛，办事不牢。他走过去一看，大吃一惊，嘴巴早已张成了 O 字形，敢情这哥俩上起网来了。

“我说你俩可够悠闲的啊，活还没干完，就先玩儿上了。”老胡显得有点儿生气。

小付呵呵一笑，“老胡，你别急，系统我俩早就装完了。我俩这不是想在点评网上看看，这附近有什么好吃的，这顿酒你是请定了。”

“好啊，你俩还真沉得住气。真的做完了？”

“这还有假？不信你看。”

老胡仔细一瞧，果然 1000 台服务器全都安装完成了。虽然输了赌局，但老胡心里却很开心。本来计划了一周的时间来搞这事儿，弄不好还得加班。没想到这两个小子一天就做完了，后面几天就等于给自己放了个小长假。想到这儿，他又有点儿发愁，“好是好，只是我安排了大家下周才做验收，后面几天你们打算怎么办呢？”

小张和小付相视一笑，“老胡你忘了赌约了么？”

“当然没有。说好的，我请你俩吃一周的饭。”

“一周的饭就算了。早就听人家说‘上有天堂、下有苏杭’，我们两个都是北方人，难得来一次杭州，想借这次机会游历一下江南美景。反正咱们三个都没什么事儿，要不然这样，请你这个杭州人给我俩做向导，吃饭喝酒我们哥俩包了。”

“好好，我愿赌服输。不过，出去游山玩水这事儿可得保密呀。”

“哈哈，那是当然。”

“不过，今天这顿我得请，怎么说我也得尽一下地主之谊。明天我带你们好好逛逛杭州城。”

---

## 7.1 高效交付解决方案如何选型

自动化系统部署并不是一个新话题。我们知道手工安装的话，执行效率主要受限于两个因素：一是手动选择浪费大量的时间；二是安装介质对复制速度的影响。

### 1. Kickstart/Cobbler

早期自动化部署基于 Kickstart+DHCP+TFTP+FileServer (HTTP/NFS/FTP) 的架构模式。Kickstart 是一个带有特定语法格式的自动应答文件，在安装操作系统之前客户端通过读取 Kickstart 配置文件来感知安装过程中的所有应答选项，并通过网络完成文件的复制。既然是通过网络进行安装，那么就需要给客户端分配 IP 地址。首先，客户端使用带有 PXE 功能的网卡向 DHCP 服务器发送请求，获得 TCP/IP 的相关配置。DHCP 在应答中指明了 TFTP Server 的地址，需要客户端到 TFTP Server 上面装载负责引导系统启动的内核与 Boot Loader。最后，客户端完成 Kickstart 文件的读取，从文件共享上面获得安装介质并开始自动化安装的工作。Kickstart 工作流程如图 7-1 所示。

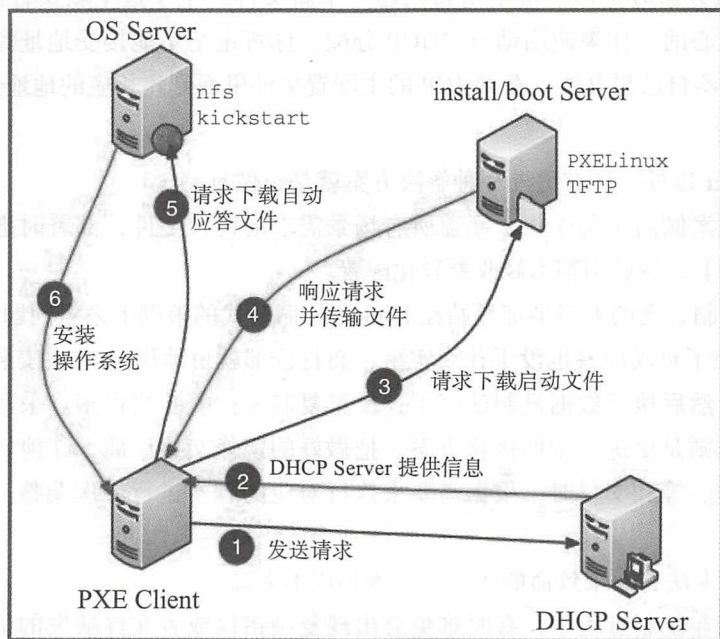


图 7-1 Kickstart 工作流程

然而，这个模式全部的精华都在于对 Kickstart 文件的控制与探索。随着对 Kickstart 文件不断地深入了解，人们对于 Kickstart 也有很多不满的地方。

首先，Kickstart 的架构过于松散，没有把这些组件进行整合；另外，Kickstart 的功能只是停留在安装选项的层面上，如果安装没有太大的改变，这个文件也没有什么发展的空

间。对于那些不在安装界面里，但是亟待解决的一些问题，Kickstart 则是将它们一股脑儿地丢给别人去完成。实在有点儿对不起自动化部署的称号，只能算是自动应答文件。

基于这些问题，一些新的解决方案就此诞生了。比较有代表性的就是自动化部署系统 Cobbler。它的核心机制依旧是 Kickstart 技术，但是它做了几件非常伟大的事情。

第一，实现了统一大业，把过去零碎的组件进行了系统整合，用户可以通过控制 Cobbler 对所有组件进行编排控制，实现了集中管理功能。除此之外，像 yum 源、配置文件、软件包、电源管理等都被纳入了 Cobbler 的管辖范围之内，基本上实现了一事不烦二主。

第二，封装了很多实用的组件功能，真正地解决了系统部署上的一些难题。例如，在一些新硬件的场景，由于原有内核不能识别，你不得不自己提前做好驱动加载的工作。而 Cobbler 却可以轻松地完成。

第三，管理的精细度增强了。过去的 Kickstart 给人有些囫囵吞枣的感觉。比如 IP 地址的部署，如果你在 Kickstart 里使用静态地址分配，则只适用于安装一台服务器。当需要部署第二台服务器时，你不得不重新创建一个新文件。至于这个新文件怎么创建出来，Kickstart 是不关心的。如果使用动态 DHCP 分配，你肯定是不能接受地址漂移的情况出现的，于是你不得不自己想办法，在 DHCP 的主配置文件里面创建相应的地址保留。

## 2. 镜像复制

除了 Cobbler 以外，还有另外一种解决方案就是镜像复制。

镜像复制方案倾向于制作一个涵盖所有场景需求的镜像文件，部署时推送这个镜像文件并解压到系统上，最后用脚本修改差异化配置。

推崇镜像复制方案的人多半都是站在 Kickstart 旧时代的浪潮上来看问题的。他们认为，既然 Kickstart 除了自动应答也没干什么实事，而且按部就班地用 yum 去安装太费劲，不如直接做成镜像，然后执行数据复制就好了，反正复制要比安装快得多。采取镜像灌装+脚本初始化的模式则是更进一步的优化方案。把做好的镜像交给厂商，让他们在出厂时帮助你把系统灌装好。等到交付时，根据需要来执行对应的脚本进行二次调整。如此一来，岂不快哉？

但是，镜像大法有一个致命的弱点——维护成本太高。

首先，镜像在复制过程中，有时难免会出现复制错误或者文件缺失的情况，这个问题却很难检查。往往是等到问题出现后才发现，但为时已晚。而以 Cobbler 为代表的 Kickstart 派系的部署方案则完全没有这个顾虑。因为系统是通过 yum 方式去安装的，最后一步会进行文件校验，这显然要比镜像复制靠谱得多。

其次，做镜像也很痛苦。应用场景不止一个，数据库和前端应用的配置肯定是不一样的，那么到底该怎么办呢？如果按照场景定制镜像，那你就需要维护多个版本，如果镜像只做一个，那就需要融合所有场景的情况，这个镜像文件对于每一个场景来说都存在着冗



余的内容。

最要命的是镜像的维护，任何一点点改动都涉及镜像文件的重新封装。尤其是在对现有模板进行软件包删除、Patch 更新或者新驱动编译的时候，烦琐的封包操作真的是非常麻烦。因为镜像修改后必须进行验证，以防止修改错误。涉及配置文件的改动也是一样，假设用脚本调整，则会出现多路径维护。最后你就会发现，有些修改在镜像里，有些则在脚本中，整个逻辑就全乱了。

Kickstart 模式则不然，维护配置文件要比维护镜像文件省心得多。Kickstart 负责配置选项、驱动、系统分区和软件包，而剩下的所有工作都交给脚本完成。脚本应当按照结构化层级进行规划，而不是使用平铺的模式。这种结构化层级的优势是：如果发生配置变更，修改内容越靠近分支，其修改成本就越低，其影响范围可控；而且不受多场景差异化的干扰，不必刻意考虑融合所带来的种种兼容性问题。

况且 Kickstart 的交付速度并不那么令人失望。我实测过，一台 Apache 可以很轻松地并发安装 200 台以上的服务器，而部署时间不到十分钟，只要硬件和网络没有问题，成功率基本上就是 100%。照此计算，单就系统部署这项工作而言，日交付量至少可以达到 5000 台以上。如果采用多个 HTTP 服务器进行分发，日交付 30 000 台是完全可行的。这足以应对绝大多数场景的需求了。

## 7.2 服务器设置详解

本书在第 3 章和第 4 章中，谈到了数据中心机柜布局和网络地址规划的重要性。建议带外地址不要采用随机分配的方式，这种方案不利于数据中心设备的日常管理。但是，静态分配会增加服务器初始化的工作量，纯手工操作的效率只有 100 台/人天，因此使用命令行接口来完成大批量修改工作是势在必行的。下面是我们实际生产环境对服务器初始化要求的范例，如表 7-1 所示。

表 7-1 服务器初始化设置范例表

| Item | Key                       | Value    | Comment |
|------|---------------------------|----------|---------|
| CPU  | Hardware Prefetcher       | Enabled  | —       |
|      | Hyper-Threading           | Enabled  | —       |
|      | QPI Speed                 | Enabled  | —       |
|      | Turbo Boost               | Enabled  | —       |
|      | Virtualization Technology | Enabled  | —       |
|      | C States                  | Disabled | —       |
|      | C1 Enhanced Halt Stat     | Disabled | —       |

(续)

| Item  | Key                          |           | Value         | Comment         |
|-------|------------------------------|-----------|---------------|-----------------|
| Boot  | Boot Mode Settings           |           | BIOS          | —               |
|       | Boot 1st Sequence            |           | Hard Disk     | —               |
|       | Boot 2nd Sequence            |           | 1st Port      | 第一个网口           |
|       | Boot Sequence Retry          |           | Enabled       | 启动失败后允许重试       |
| PXE   | 1st Port Network Adapter     |           | Enabled       | 第一个网口启用 PXE     |
|       | Other Network Adapters       |           | Disabled      | 禁用 PXE 而非设备     |
| OOB   | User                         |           | ***           | 初始化管理员用户名       |
|       | Password                     |           | ***           | 初始化管理员密码        |
|       | IPMI over LAN                |           | Enabled       | 支持 IPMI 协议      |
|       | 1st Virtual Console ( 5900 ) |           | Enabled       | B/S 控制台         |
|       | 2nd Virtual Console ( 5901 ) |           | Enabled       | C/S 控制台         |
|       | SSH Service                  |           | Enabled       | 支持 SSH 访问       |
|       | HTTP/HTTPS Service           |           | Enabled       | 支持 B/S 访问       |
|       | E-mail Service               |           | Enabled       | 启用邮件告警功能        |
|       | FTP Service                  |           | Disabled      | 如有需禁用           |
|       | SNMP Service                 |           | Disabled      | 如有需禁用           |
|       | Telnet Service               |           | Disabled      | 如有需禁用           |
| Log   | Audit Log                    |           | Enabled       | 原则上启用所有日志       |
|       | Power Log                    |           | Enabled       | 原则上启用所有日志       |
|       | System Event Log             |           | Enabled       | 原则上启用所有日志       |
| Alert | Facility                     | HW Status | Enabled       | 硬件运行状态事件告警      |
|       |                              | Storage   | Enabled       | 存储事件告警          |
|       |                              | Other     | Disabled      | —               |
|       | Level                        | error     | Enabled       | 开启 warning 以上级别 |
|       |                              | Warning   | Enabled       | 开启 warning 以上级别 |
|       |                              | Info      | Disabled      | —               |
| Power | Power Policy Mode            |           | Not Redundant | 双路供电策略          |
|       | Power Hotspare               |           | Enabled       | 打开电源冗余功能        |

修改这些服务器设置的方法大体上可以分为两种。一种是使用 IPMI 来定义诸如带外管理地址、登录用户等初始化信息；另一种是使用厂商提供的私有化接口设置，诸如 BIOS、电源策略、日志告警等内容。



## 7.2.1 IPMI

关于 IPMI，我们在第 5 章已经做了详细介绍，请读者自行参阅相关部分。这里主要是完成两件事情：初始化带外管理地址并设置用户密码。我在这里以 DELL 的 Rack 系列服务器为例，给大家介绍一下如何完成相关工作。

首先，设置带外地址可用。注意：修改网关之前，一定要先改掩码，如果网关不在掩码的范围内，会引发修改失败的错误。另外，192.168.0.120 是 DELL 服务器出厂设置的默认地址，只能接一台改一台。建议你准备一根长网线，然后将笔记本放置在机房的正中间操作，找个小伙伴在另一端帮你插网线，效率会快很多。

```
# ipmitool -I lanplus -H 192.168.0.120 -U <USER> -P <PASSWORD> lan set <CHANNEL>  
netmask <NETMASK>  
# ipmitool -I lanplus -H 192.168.0.120 -U <USER> -P <PASSWORD> lan set <CHANNEL>  
defgw ipaddr <GATEWAY>  
# ipmitool -I lanplus -H 192.168.0.120 -U <USER> -P <PASSWORD> lan set <CHANNEL>  
ipaddr <IPADDR>
```

接下来，是修改带外管理的用户名和密码。

```
# ipmitool -I lanplus -H <IPADDR> -U <USER> -P <PASSWORD> user set name <USER_  
ID> <USER_NAME>  
# ipmitool -I lanplus -H <IPADDR> -U <USER> -P <PASSWORD> user set password  
<USER_ID> <NEW_PASSWORD>
```

如果执行 ipmitool 命令失败并返回 Unable to establish LAN session，这里会有两种可能的原因。第一种原因是带外管理卡的问题。如果反复插拔网线不起效，可以关闭电源并拔掉电源线，然后按住电源按钮至少 30s 进行放电操作。第二种原因是 IPMI Over LAN 未启用造成的。执行 racadm set iDRAC.IPMILan.Enable Enabled 即可启用 IPMI Over LAN。关于 racadm，我会在下一节做详细介绍。

## 7.2.2 racadmin

racadmin 全称是远程访问卡管理，它是 DELL 的一个私有化的管理工具，用于管理 DELL 服务器的各项设置。可以说，只要能在管理界面上看到的，就可以通过 racadmin 来修改。有些读者可能会有这样的疑问，为什么一定要使用 racadmin，而不是在图形下操作呢？因为图形操作实在是太慢了。像 BIOS 的一些设置，如果用 F2 键引导，修改完成后再去重启，少说也得花个五六分钟。若是一口气改个两三百台，我估计换谁都得吐了。

设置好带外管理地址后，可以直接通过 SSH 登录 DELL 的 racadm 接口，执行 racadm 的子命令 set 修改配置项，子命令 get 用于查询配置项的值。所有的配置项是一个树形的结构，层级之间使用“.”来分隔。有些配置设置后不会立即生效，需要重启服务器，例如 BIOS 中的很多设置。因此，需要使用子命令 jobqueue create 创建一个计划任务，当服务器



重启时会自动执行这些任务，以达到更新的效果。

以下这些内容均是按照表 7-1 的要求执行的命令，请读者自行对照参考。

### (1) CPU 特性通用设置

```
racadm set BIOS.ProcSettings.ProcHwPrefetcher Enabled
racadm set BIOS.ProcSettings.ProcVirtualization Enabled
racadm set BIOS.ProcSettings.QpiSpeed MaxDataRate
racadm set BIOS.SysProfileSettings.SysProfile Custom
racadm set BIOS.SysProfileSettings.ProcTurboMode Enabled
racadm get BIOS.SysProfileSettings.ProcCStates Disabled
racadm get BIOS.SysProfileSettings.ProcC1E Disabled
racadm jobqueue create BIOS.Setup.1-1
```

### (2) BIOS 模式及启动顺序调整

```
racadm set BIOS.BiosBootSettings.BootMode Bios
racadm set BIOS.BiosBootSettings.BootSeq \
"HardDisk.List.1-1,NIC.Integrated.1-1-1"
racadm set BIOS.BiosBootSettings.BootSeqRetry Enabled
racadm jobqueue create BIOS.Setup.1-1
```

### (3) 将第一块网卡设置为 PXE 启动

```
racadm set NIC.NICConfig.1.LegacyBootProto PXE
racadm set NIC.NICConfig.2.LegacyBootProto NONE
racadm set NIC.NICConfig.3.LegacyBootProto NONE
racadm set NIC.NICConfig.4.LegacyBootProto NONE
racadm jobqueue create NIC.Integrated.1-1-1
racadm jobqueue create NIC.Integrated.1-2-1
racadm jobqueue create NIC.Integrated.1-3-1
racadm jobqueue create NIC.Integrated.1-4-1
```

### (4) 带外管理设置

```
// 启用 IPMI
racadm set iDRAC.IPMILan.Enable Enabled
// 启用并配置虚拟控制台
racadm set iDRAC.VirtualConsole.Enable Enable
racadm set iDRAC.VirtualConsole.Port 5900
racadm set iDRAC.VirtualConsole.Timeout 1800
// 启用并配置 VNC 服务
racadm set iDRAC.VNCServer.Enable Enabled
racadm set iDRAC.VNCServer.Password <PASSWORD>
racadm set iDRAC.VNCServer.Port 5901
racadm set iDRAC.VNCServer.Timeout 300
// 启用并配置 SSH 服务
racadm set iDRAC.SSH.Enable Enabled
racadm set iDRAC.SSH.Port 22
racadm set iDRAC.SSH.Timeout 1800
// 启用并配置 WebServer 服务
racadm set iDRAC.WebServer.Enable Enabled
```

```

racadm set iDRAC.WebServer.HttpPort 80
racadm set iDRAC.WebServer.HttpsPort 443
racadm set iDRAC.WebServer.HttpsRedirection Enabled
racadm set iDRAC.WebServer.Timeout 1800
// 启用并配置 E-mail 告警
racadm set iDRAC.EmailAlert.1.Enable Enabled
racadm set iDRAC.EmailAlert.1.Address <Receiver E-MAIL_ADDRESS>
racadm set iDRAC.EmailAlert.1.CustomMsg <MESSAGE>
racadm set iDRAC.RemoteHosts.SMTPServerIPAddress <SERVER_ADDRESS>
// 禁用 SNMP 和 Telnet
racadm set iDRAC.SNMP.AgentEnable Disabled
racadm set iDRAC.Telnet.Enable Disabled

```

### (5) 启用并配置日志功能

```

racadm set iDRAC.Syslog.PowerLogEnable Enabled
racadm set iDRAC.Syslog.SysLogEnable Enabled
racadm set iDRAC.Syslog.Port 514
racadm set iDRAC.Syslog.Server1 <SERVER_ADDRESS>

```

### (6) 告警信息的配置

```

// 启用告警
racadm set iDRAC.IPMILan.AlertEnable 1
// 设置发送邮件账户及域, 启用 critical 级别的告警
racadm set iDRAC.NIC.DNSRacName <SENDER_ACCOUNT>
racadm set iDRAC.NIC.Static.DNSDomainName <SENDER_DOMAIN>
racadm eventfilters set -c iDRAC.alert.system.critical -a None -n Email
racadm eventfilters set -c iDRAC.alert.storage.critical -a None -n Email

```

### (7) 调整电源策略为非冗余模式

```

racadm set System.Power.RedundancyPolicy 'Not Redundant'
racadm set System.Power.HotSpare.Enable Enabled

```

## 7.2.3 SMASH CLP

和 racadmin 类似, HP、H3C 等使用的是 SMASH CLP。它们的设计结构和语法都是相似的。我们依旧以表 7-1 为例, 给出 SMASH CLP 的一些命令示例。

### (1) BIOS 模式及启动顺序调整

```

set /system1/bootconfig1 oemhp_bootmode=Legacy
set /system1/bootconfig1/bootsource1 bootorder=1
set /system1/bootconfig1/bootsource1 bootdevice=BootFmDisk
set /system1/bootconfig1/bootsource2 bootorder=2
set /system1/bootconfig1/bootsource2 bootdevice=BootFmNetwork1

```

### (2) 带外管理设置

```

// 设置相关服务端口, 会自动重启 iLO
set /map1/config1 oemhp_rcport=5900
set /map1/config1 oemhp_sshport=22

```



```
set /map1/config1 oemhp_httpport=80
set /map1/config1 oemhp_sslport=8443
// 禁用 SNMP
set /map1/config1 oemhp_snmp_access=no
```

### (3) 告警信息的配置

```
// 启用并配置 E-mail 告警
set /map1/oemhp_alertmail1 oemhp_alertmail_smtp_server=<ADDRESS>
set /map1/oemhp_alertmail1 oemhp_alertmail_smtp_port=25
set /map1/oemhp_alertmail1 oemhp_alertmail_enable=yes
// 设置发送邮件账户及域, 会重启 iLO
set /map1/oemhp_alertmail1 \
oemhp_alertmail_email=<SENDER_E-MAIL_ADDRESS>
set /map1/oemhp_alertmail1 \
oemhp_alertmail_sender_domain=<SENDER_DOMAIN>
set /map1/enetport1 SystemName=<SENDER_ACCOUNT>
set /map1/dnsendpt1 DomainName=<SENDER_DOMAIN>
```

### (4) 启用并配置日志功能

```
set /map1/oemhp_ahs1 EnabledState=yes
set /map1/oemhp_syslog1 oemhp_syslog_serveraddress=<IPADDRESS>
set /map1/oemhp_syslog1 oemhp_syslog_port=514
set /map1/oemhp_syslog1 oemhp_syslog_enable=yes
```

## 7.3 Cobbler 部署系统详解

这一节, 我们先从 Cobbler 的架构入手, 让读者理解各个组件的含义和它们之间的关系。然后, 给读者一个简单明了的部署方法, 读者根据实际情况略作修改后, 就可以顺利完成相关配置。笔者会根据自己的实际环境, 提供一个实例化的命名范例, 并把部署过程中出现的各种问题的排错方法公布出来, 帮助读者避开那些已知的陷阱。

### 7.3.1 理解 Cobbler 架构

图 7-2 所示为 Cobbler 组件架构图, 该图对我们理解 Cobbler 架构有很大的帮助。

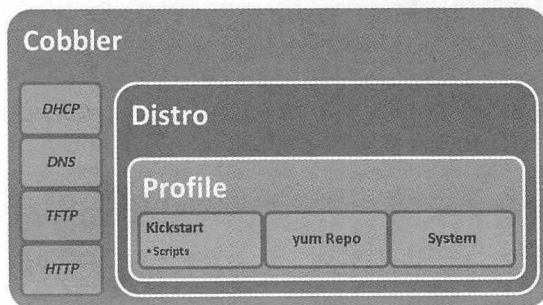


图 7-2 Cobbler 组件架构图



Distro 可以看作操作系统的发行版本，你的生产环境中涉及几种操作系统，就需要创建几个 Distro。请注意，Distro 不仅仅指操作系统种类，同时也包括版本。即便是不同版本之间的 CentOS，也将被看成不同的 Distro。假设生产环境需要 CentOS 5、CentOS 6 和 RHEL 7 三个版本，那么就需要创建三个 Distro。

我们可以把 Profile 当作一个部署模板，你的生产环境中几种应用场景就需要几个 Profile。例如，一个数据中心当中至少应该包括数据库、前端应用、基础管理设备，很显然它们的安装要求是不一样的，所以你需要创建三个 Profile。如果前端应用还涉及虚拟机的应用，那么虚拟机和宿主机的分区信息和物理服务器还会有所区别，则应该再增加两个 Profile。

System 是 Profile 的一个部署实例，它包含了一台实际需要部署的服务器的相关系统信息，这些信息包括主机名、TCP/IP 设置、网卡 Bonding 模式、VLAN 配置、使用哪个操作系统（由 Distro 决定）、使用哪些 yum 源（由 yum repo 决定）以及部署成什么样子（由 yum repo 决定）等。如果你安装需要部署 100 个系统，那么就需要创建 100 个 System。

### 7.3.2 Cobbler 的安装配置

这一节将为大家介绍如何完成 Cobbler 的安装配置工作。整个过程分为六个步骤。

#### 1. 安装

根据 Cobbler 的架构图已知，我们需要安装 DHCP、HTTP、TFTP 等服务组件。除此之外，还需要 rsync 和 xinetd 的支持。rsync 用于同步配置文件，xinetd 是 TFTP 和 rsync 的依赖服务。请使用 yum 命令完成软件包的安装。

```
# yum install -y cobbler dhcp httpd xinetd rsync tftp
```

注意：在大多数的场景中，Cobbler 部署在非 SELinux 的环境下。如果你的生产环境启用了 SELinux，还需要完成一些额外的工作，详细解决方案请参见如下链接。

<https://github.com/cobbler/cobbler/wiki/SELinux>

#### 2. 配置主配置文件

假设 Cobbler 服务器的地址为 192.168.0.100。我们通过修改其主配置文件 /etc/cobbler/settings，实现让 Cobbler 接管 DHCP 等所有的服务。Cobbler 默认对 TFTP 的管理是启用的，但其他服务处于禁用模式。执行如下命令，指定 Cobbler 的主机地址，并启用 DHCP 和 rsync。

```
# sed -e "s/server: 127.0.0.1/server: 192.168.0.100/" -e "s/dhcp: 0/dhcp: 1/" -e  
"s/rsync: 0/rsync: 1/" /etc/cobbler/settings
```

我们用 Cobbler 自动装好系统后，系统的 root 密码本应是在 Kickstart 的文件中指定。如果 Kickstart 里没有写，Cobbler 会用自己内置的默认密码来替换。接下来我们就要配置这

个默认密码。默认密码的配置项叫 `default_password_crypted`，它的值是一个哈希值。这里我们假定密码为 `password`，如何将它转换成对应的哈希值呢？可以通过执行如下这条命令来实现。

```
[root@station103 ~]# python -c "import crypt,getpass;print crypt.crypt('password',crypt.mksalt(method=crypt.METHOD_SHA256))"
$5$7gfQWf.ZQVQEnsPf$FVbG5E9m0c2ORVid3ClS2.trEGt3HMfnM22.LsfoEH.
```

得到哈希值后，我们用它替换掉 `default_password_crypted` 条目中原有的数值。

```
default_password_crypted: "$5$7gfQWf.ZQVQEnsPf$FVbG5E9m0c2ORVid3ClS2.trEGt3HMfnM22.LsfoEH."
```

### 3. 配置安装密码

设置安装密码主要用于防止未授权使用 Cobbler。如果没有这个密码，任何人在机房内，都可以通过 PXE 请求安装一套线上系统。例如，我在笔记本上起一台虚拟机发送 PXE 请求，线上配置就搞到手了。这是非常危险的，因此必须设置安装密码，防止配置泄露。

设置完成后，在正式部署系统前，Cobbler 会请求先输入安装密码，否则会拒绝后续的执行。注意，这个密码校验是防止非授权安装的。对于那些已经加入 System 实例或使用 koan 主动发起部署请求的主机，不会受到影响。

实现这个配置，需要修改两个地方。

首先，修改配置文件 `/etc/cobbler/pxe/pxedefault.template`，在 MENU TITLE Cobbler 下面添加如下内容。最后的 Hash 字符串，可使用命令 `openssl passwd -1` 来生成。

```
MENU MASTER PASSWD $1$LooZh28v$Xv0Bgrr8Bqo63wwv6prgE0
```

然后，修改配置文件 `/etc/cobbler/pxe/pxepfile.template`，在 `$menu_label` 下面添加如下内容。

```
MENU PASSWD
```

### 4. 配置 DHCP

假设我们需要部署系统的地址属于 192.168.1.0/24 这个网段，网关是 192.168.1.254，地址池是 192.168.1.1-192.168.1.200，可按照如下配置修改。

```
修改 /etc/cobbler/dhcp.template
subnet 192.168.1.0 netmask 255.255.255.0 {
    option routers                192.168.1.254;
    option domain-name-servers   202.106.0.20;
    option subnet-mask           255.255.255.0;
    range dynamic-bootp          192.168.1.1 192.168.1.200;
    default-lease-time            21600;
    max-lease-time                43200;
    next-server                   $next_server;
    class "pxeclients" {
        match if substring (option vendor-class-identifier, 0, 9) = "PXE-
```

```

        Client";
    if option pxe-system-type = 00:02 {
        filename "ia64/elilo.efi";
    } else if option pxe-system-type = 00:06 {
        filename "grub/grub-x86.efi";
    } else if option pxe-system-type = 00:07 {
        filename "grub/grub-x86_64.efi";
    } else {
        filename "pxelinux.0";
    }
}
}

```

注意：在修改时，我们需要修改 `/etc/cobbler/dhcp.template`，而不是 DHCP 原有的主配置文件 `/etc/dhcp/dhcpd.conf`。因为 cobbler 已经接管了 DHCP 服务，在执行 `cobbler sync` 同步的时候，将会复制 `dhcp.template` 替换掉 `dhcpd.conf`。

## 5. 下载 PXE 引导文件

执行命令 `cobbler get-loaders`，完成引导文件的下载。当下载完成后，应当能在目录 `/var/lib/cobbler/loaders/` 下看到全部的 loader 文件，如下所示。

```

[root@station100 ~]# ls /var/lib/cobbler/loaders/
COPYING.elilo      COPYING.yaboot    grub-x86_64.efi  menu.c32         README
COPYING.syslinux  elilo-ia64.efi   grub-x86.efi     pxelinux.0       yaboot

```

我在使用 2.6.3 版本的 Cobbler 尝试执行这条命令时出现了错误，原因是该版本的下载链接为 <http://www.cobblerd.org/loaders/>，可下面根本就没有任何东西。2.4.4 版本的下载地址为 <http://dgoodwin.fedorapeople.org/loaders/>，我们可以去这里下载 loader，并完成文件名的修改。get-loaders 的全过程就是将 loader 文件下载并改名而已。你可以通过手动来完成这个操作。

## 6. 启动 cobbler 并修正相关问题

### 1) 启动 cobbler 及相关服务。

```

# service cobblerd start
# service httpd start
# service xinetd start
# chkconfig cobblerd on
# chkconfig httpd on
# chkconfig xinetd on
# chkconfig rsync on
# chkconfig tftp on
# chkconfig dhcpd on

```

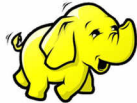
2) 修正其他问题。到此为止，Cobbler 就基本配置完毕了，我们可以执行命令 `cobbler check` 来检查一下，发现依然存在一些问题需要处理。

```

1 : ksvalidator was not found, install pykickstart
2 : fencing tools were not found, and are required to use the (optional) power

```





```
management features. install cman or fence-agents to use them
3 : since iptables may be running, ensure 69, 80/443, and 25151 are unblocked
4 : debmirror package is not installed, it will be required to manage debian
  deployments and repositories
```

pykickstart 提供了用于检验 Kickstart 语法的工具 ksvalidator，这个软件包必须安装。如果你要使用 Cobbler 来管理电源（这个不是必需的），则还需要安装 cman 或者 fence-agents。推荐你安装 fence-agent，因为它比 cman 要稍微小一些。

```
# yum install -y pykickstart fence-agents
```

debmirror 用于管理 Debian 系统部署，如果没有必要可以忽略不计。安装 debmirror 之后再执行 cobbler check 又会出现如下提示。

```
1 : comment out 'dists' on /etc/debmirror.conf for proper debian support
2 : comment out 'arches' on /etc/debmirror.conf for proper debian support
```

我们可以根据提示完成修改。

```
# sed -i -e "s/@dists/# @dists/" \
-e "s/@arches/# @arches/" \
/etc/debmirror.conf
```

3) 再次执行 cobbler check，正常情况下应当是如下状态。

```
[root@station100 kickstarts]# cobbler check
No configuration problems found. All systems go.
```

### 7.3.3 命名规范

规范的命名有助于日后的方便管理，如果你不打算给别人添麻烦，最好在创建系统之初做好命名规范工作。下面是笔者创建的一些范例，仅供读者朋友们做参考。

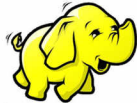
#### 1. 目录结构定义范例

目录结构尽可能采取层级嵌套的方式，这符合现代系统管理的主流风格，比起把所有内容平铺到一个目录当中的做法，层级嵌套的效果显然要好很多。

这个范例将所有资源放置到一个名为 /export 的独立分区下，下面所示的内容便是使用 tree 命令后展示出来的效果。

```
-----
Directory named "/export" structure
-----
```

```
/export          // 用于存放软件资源的总目录
|-- clamav       // 用于存放软件 clamav (epel 中没有的版本)
|-- clamav-db    // 用于存放病毒库文件，用于内网病毒库升级
|-- epel         // 用于存放额外软件包
|-- isos         // 用于存放系统镜像
`-- os           // 由系统镜像挂载的系统安装目录
    |-- centos6.5
```



```
|    |-- x86_64
|-- rhel6.5
|    |-- x86_64

-----
Directory named "/opt" structure
-----

/opt                                // 用于存放配置文档及执行脚本的总目录
|-- conf                            // 用于存放配置文件的总目录
|   |-- aide
|   |-- clamav
|   |-- cobbler
|   |-- puppet
|   |-- salt
|   |-- zabbix
|-- scripts                         // 用于存放执行脚本的总目录
|   |-- manage                     // 管理脚本——后期进行维护时有可能需要用到
|   |-- post                       // 安装脚本——主要用来对应 kickstart 文件
|   |-- post.d                     // 安装配置脚本——post 脚本中所对应的具体配置
|-- view                           // 用于脚本只读视图的总目录（仅供浏览器直接查看）
|   |-- manage
|   |-- post
|   |-- post.d
```

## 2. Distro 命名规范定义范例

下面是我在实际生产环境中对 Distro 所采用的命名范例。

发行产品版本 - 架构 (%Breed%%Release%-%Arch%)

命名示例: CentOS 6.5-x86\_64

## 3. Profile 命名规范定义范例

Profile 命名有以下几项需要规范定义:

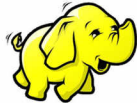
- ☐ Mgmt——用于管理区的主机;
- ☐ DB——用于数据库区的主机;
- ☐ App——用于 DMZ 区的主机;
- ☐ Hadoop——用于大数据区的主机。

## 4. 文件定义范例

下面是我在实际生产环境中对 Profile、Kickstart 和各个脚本所采用的命名范例, 如表 7-2 所示。

表 7-2 文件命名定义范例表

| Profile            | Kickstart        | Post Script      | Description |
|--------------------|------------------|------------------|-------------|
| Mgmt-ExternalDNS-M | ExternalDNS-M.ks | ExternalDNS-M.sh | 外网主 DNS     |
| Mgmt-ExternalDNS-S | ExternalDNS-S.ks | ExternalDNS-S.sh | 外网备 DNS     |



(续)

| Profile            | Kickstart        | Post Script      | Description |
|--------------------|------------------|------------------|-------------|
| Mgmt-InternalDNS-M | InternalDNS-M.ks | InternalDNS-M.sh | 内网主 DNS     |
| Mgmt-InternalDNS-S | InternalDNS-S.ks | InternalDNS-S.sh | 内网从 DNS     |
| Mgmt-Saltstack     | Saltstack.ks     | Saltstack.sh     | 管理主机        |
| Mgmt-Zabbix        | Zabbix.ks        | Zabbix.sh        | 监控主机        |
| DB-oracle          | oracle.ks        | oracle.sh        | Oracle 数据库  |
| DB-mysql           | mysql.ks         | mysql.sh         | MySQL 数据库   |
| App-host           | host.ks          | host.sh          | 应用物理机       |
| App-vmhost         | vmhost.ks        | vmhost.sh        | 应用宿主机       |
| App-vm             | vm.ks            | vm.sh            | 应用虚拟机       |
| Hadoop-nnodes      | nnodes.ks        | nnodes.sh        | 名称节点        |
| Hadoop-dnodes      | dnodes.ks        | dnodes.sh        | 数据节点        |

5. Cobbler 元素关系定义范例

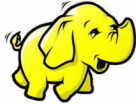
我来讲解一下我的设计思路。对于部署架构的规划，我会根据现有业务场景的需要，针对不同场景定制不同的 Distro 和 Profile，每个 Profile 对应自己的 Kickstart。例如，我有 MySQL 和 Web Server 两种部署方案，首先做一个最小化系统安装的模板——名为 Base 的 Profile 和名为 base.ks 的 Kickstart。在 base.ks 的基础上添加 MySQL 需要安装的软件包并修改分区设置，并为此创建了名为 MySQL 的 Profile 和名为 mysql.ks 的 Kickstart，Web Server 的同理。我希望 Kickstart 文件只描述两个内容——分区设置与安装所需的软件包，其他的调整均通过脚本实现。

关于 %post 脚本的设计，我仅使用 curl 调用的方式来实现。首先在 mysql.ks 里利用 curl 调用执行脚本 mysql.sh，而 mysql.sh 里也不写任何具体的内容，依旧利用 curl 调用二级脚本 Detail01、Detail02、Detail03 等。也就是说，每一个 Profile 都对应一个 Kickstart，每一个 Kickstart 对应一个执行脚本，执行脚本又对应若干二级脚本，而这些二级脚本才是真正用来执行配置修改任务的。我把所有的执行脚本放置到一个名为 /opt/scripts/post/ 的子目录中，把所有的二级脚本放置到一个名为 /opt/scripts/post.d/ 的子目录中。相信大家也看懂了，这种方式其实就是效仿系统 /etc/rcX.d/ 和 /etc/init.d/ 的关系而建立的。这样做的好处在于：当需要调整配置的时候，可以直接修改二级脚本，然后执行一下就能看到效果，而不是重新“kick”一遍主机。

7.3.4 创建资源目录

前面我们已经给出了相关的命名规范，读者朋友可根据自身情况创建对应的资源目录。目录构建完成后，至少需要下载 CentOS 的系统 IOS 和 EPEL 源，这是安装系统和常用工具





所必需的软件仓库。CentOS 官网在世界各地都有镜像站点，推荐大家去国内的站点下载。这样在网速和稳定性上都能有所保证。但那只能获取 CentOS 最新版本的镜像文件，如果你需要老版本的镜像文件，需要到 <http://vault.centos.org> 去下载。

下载 EPEL 软件包的时候，需要注意一下。因为 EPEL 目录下包含了非常多的 RPM 包，wget 时需要直接下载它的上级目录。在这个过程中，wget 会产生一些无用的日志和空目录，建议下载完成后清理一下。

举个例子，假设我要在 /export/epel/ 目录下放置 EPEL 的仓库，使用命令 wget 在当前目录位置下载。注意，因为下载的时间比较长，建议使用 nohup 命令防止链路意外中断。

```
# cd /export/epel
# nohup wget -c -m -np http://mirrors.hustunique.com/epel/6Server/x86_64/ &
```

待下载完成后，清除无用的文件。

```
# mv /export/epel/mirrors.hustunique.com/epel/6Server /export/epel/
# rm -rf /export/epel/mirrors.hustunique.com /export/epel/nohup.out
# find 6Server/ -name "index.html" -exec rm -f {} \;
```

### 7.3.5 创建 Cobbler 部署模板与实例

#### 1. 创建 Distro

执行如下命令完成 Distro 的创建。

```
# cobbler distro add \
--name=centos6.5-x86_64 \
--kernel=/var/www/html/centos6.5-x86_64/isolinux/vmlinuz \
--initrd=/var/www/html/centos6.5-x86_64/isolinux/initrd.img \
--arch=x86_64 \
--breed=redhat \
--ksmeta="directory=http://@@http_server@@/centos6.5-x86_64"
```

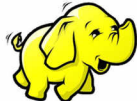
#### 2. 修改 Kickstart 文件

Kickstart 文件包含如下几类：

- ☐ Installation Options;
- ☐ Package Selection;
- ☐ Pre-installation Script;
- ☐ Post-installation Script.

Installation Options 是图形安装中手动选择的一些项目，Package Selection 是我们需要安装的软件包。Pre-installation Scripts 负责完成安装之前的工作，这里重点是 RAID 的配置。Post-installation Script 负责完成安装之后的工作，可以把差异化的修改全部放置在这里执行。

但是，我不建议把具体内容全部平铺在 kickstart 里面，而是建议采用嵌套的方式。Cobbler 的示例模板中提供了 \$SNIPPET() 的功能，我们也可以照此效法。



### 注意：

Kickstart 模板由 Cobbler 提供，这里不再赘述具体的内容，Kickstart 文件的语法详解请读者参考如下链接：[https://www.centos.org/docs/5/html/Installation\\_Guide-en-US/ch-kickstart2.html](https://www.centos.org/docs/5/html/Installation_Guide-en-US/ch-kickstart2.html)。

## 3. 创建 Repo

执行如下命令完成 Repo 的创建。

```
# cobbler repo add --name=CentOSRepo \  
--mirror=http://192.168.0.100/centos6.4-x86_64/ --mirror-locally=N  
# cobbler repo add --name=EPELRepo \  
--mirror=http://192.168.0.100/epel/6Server/x86_64/ --mirror-locally=N  
# cobbler reposync
```

--mirror-locally=N 代表只添加 repo 的配置文件而不再复制 repo 仓库的软件包。因为我们已经把 repo 的仓库存储在了 HTTP 的主目录，所以一定要添加这个参数。否则，默认会把所有的软件都复制到 /var/www/cobbler/repo\_mirror/ 目录下。

## 4. 创建 Profile

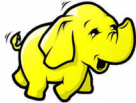
执行如下命令完成 Profile 的创建。

```
# cobbler profile add \  
--name=centos6.5-x86_64 \  
--distro=centos6.5-x86_64 \  
--kickstart=/var/lib/cobbler/kickstarts/sample.ks \  
--repos="CentOSRepo EPELRepo"
```

## 5. 创建 System

执行如下命令完成 System 的创建。

```
# Add a system and bond master into cobbler.  
cobbler system add --name=<NAME> --hostname=<HOSTNAME> \  
--interface=<BOND_NIC> --interface-type=bond \  
--bonding-opts="miimon=100 mode=4 xmit_hash_policy=layer2+3" \  
--ip-address=<BOND_IP> --subnet=<NETMASK> --gateway=<GATEWAY> --static=1 \  
--profile=<PROFILE>  
  
# Append slave A to bond master.  
cobbler system edit --name=<NAME> --mac==<MAC> \  
--interface=<SLAVE1_NIC> --interface-type=bond_slave \  
--interface-master=<BOND_NIC> \  
--profile=<PROFILE>  
  
# Append slave B to bond master.  
cobbler system edit --name=<NAME> --mac==<MAC> \  
--interface=<SLAVE2_NIC> --interface-type=bond_slave \  
--interface-master=<BOND_NIC> \  
--profile=<PROFILE>
```



### 7.3.6 Cobbler 里面出现的坑

在历经过上万台系统部署之后，笔者总结了不少排障的经验，在这里一并分享出来供广大读者参考。

#### 1. DHCP 客户端获取 IP 地址失败的场景

这一类问题基本都出现在跨网段的场景，故障界面如图 7-3 所示。

```
Scanning for devices. Please wait, this may take several minutes...

Intel(R) Boot Agent GE v1.4.03
Copyright (C) 1997-2012, Intel Corporation

CLIENT MAC ADDR: EC F4 BB C3 3C E0  GUID: 44454C4C 5800 1037 8051 B3C04F573232
PXE-E51: No DHCP or proxyDHCP offers were received.

PXE-M0F: Exiting Intel Boot Agent.
No operating system is currently installed on this computer.
```

图 7-3 DHCP 客户端获取 IP 地址失败

出现 DHCP 的故障无外乎是以下几种情况，大家可以根据这些问题自检一下。

- ❑ 某些交换机有可能需要禁用 STP，否则会收不到 DHCP 请求。
- ❑ 某些交换机对 VRRP 的支持不够好，网关不接收广播报文，尝试禁用或使用 HSRP。
- ❑ 交换机没有配置 DHCP 中继的功能。
- ❑ 网卡模块损坏、网线损坏、插错网线等，这类问题的错误和截图有所不同，它会导致 PXE 在启动时提示 media failure。

除此之外，还有一种情况，由于地址资源池中已经没有可用的地址了。我们需要执行如下命令，通过清除租赁关系来释放资源。

```
# rm -f /var/lib/dhcpd/dhcpd.leases*  && /etc/init.d/dhcpd restart
```

#### 2. 未开启 ARP 转发导致的 TFTP 加载失败

没有开启 ARP 代理，在访问 TFTP 服务时会提示 ARP timeout，如图 7-4 所示。

```
Copyright (C) 1997-2012, Intel Corporation

CLIENT MAC ADDR: EC F4 BB C3 0C B4  GUID: 44454C4C 3300 104A 8034 C8C04F383232
CLIENT IP: [REDACTED]  MASK: [REDACTED]  DHCP IP: [REDACTED]
PXE-E11: ARP timeout
PXE-M0F: Exiting Intel Boot Agent.

Intel(R) Boot Agent GE v1.4.03
Copyright (C) 1997-2012, Intel Corporation

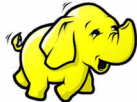
CLIENT MAC ADDR: EC F4 BB C3 0C B4  GUID: 44454C4C 3300 104A 8034 C8C04F383232
CLIENT IP: [REDACTED]  MASK: [REDACTED]  DHCP IP: [REDACTED]
PXE-E11: ARP timeout
PXE-M0F: Exiting Intel Boot Agent.

No boot device available.
Current boot mode is set to BIOS.
Please ensure compatible bootable media is available.
Use the system setup program to change the boot mode as needed.

Strike F1 to retry boot, F2 for system setup, F11 for BIOS boot manager.
```

图 7-4 未开启 ARP 转发导致的 TFTP 加载失败





### 3. 部署流程错误导致的 TFTP 加载失败

这是我在部署一台物理服务器时遇到的错误。TFTP 刚刚完成连接，很快又一下子退出了，如图 7-5 所示。

```
Strike F1 to retry boot, F2 for system setup, F11 for BIOS boot manager.  
Intel(R) Boot Agent GE v1.5.56  
Copyright (C) 1997-2014, Intel Corporation  
  
CLIENT MAC ADDR: EC F4 BB C3 C0 DC GUID: 44454C4C 3600 104B 8059 C8C04F5A3232  
CLIENT IP: [REDACTED] MASK: [REDACTED] DHCP IP: [REDACTED]  
GATEWAY IP: [REDACTED]  
TFTP.
```

图 7-5 部署流程错误导致的 TFTP 加载失败

通过使用 tcpdump 抓包来进行故障分析，结果我在里面看到了这样一行——缺少 filename "undionly.kpxe"。这个 undionly.kpxe 看着非常眼熟，它不正是配置文件 /etc/cobbler/dhcp.template 中的 if \$iface.enable\_gpxe 部分么？原来是某个同学为了测试用 Cobbler 部署虚拟机而修改了全局配置文件，他打开了 gpxe 的功能，测试完毕后却忘记关掉了。所以，只要我们改回去就可以了，修改 /etc/cobbler/setting 中的 enable\_gpxe，将 Value 设置成 0。但是这样不算完，因为 system 已经生成了，所以需要更新 system 配置或者重建 system。

### 4. loaders 文件错误导致的 TFTP 加载失败

这个问题是我在部署一个新的 Cobbler 服务器时遇到的，如图 7-6 所示。Cobbler 的版本是 2.6.3，在执行 get-loaders 的时候，Cobbler 报告 loader 文件无法下载。这个问题我们在 7.3.2 节提到过，原因是这个版本指向的 loader 下载地址是错误的。由于我手上有一份备份，对此也没在意，直接把备份文件给复制过去了。没想到就是这不经意的一个复制，竟然就引发了这个问题。

```
Intel(R) Boot Agent GE v1.5.56  
Copyright (C) 1997-2014, Intel Corporation  
  
CLIENT MAC ADDR: EC F4 BB C3 C0 DC GUID: 44454C4C 3600 104B 8059 C8C04F5A3232  
CLIENT IP: [REDACTED] MASK: [REDACTED] DHCP IP: [REDACTED]  
GATEWAY IP: [REDACTED]  
PXE-E32: TFTP open timeout  
PXE-M0F: Exiting Intel Boot Agent.  
  
No boot device available.  
Current boot mode is set to BIOS.  
Please ensure compatible bootable media is available.  
Use the system setup program to change the boot mode as needed.  
  
Strike F1 to retry boot, F2 for system setup, F11 for BIOS boot manager.
```

图 7-6 loaders 文件错误导致的 TFTP 加载失败

通过 tcpdump 抓包没有太多的收获，我只看到了一个 Extra Error，再往后面就语焉不详了。于是，我冷静下来反思系统安装的流程。当想到 TFTP 本来就是要提供 loader 文件时，自己一下子就释然了。使用 md5sum 命令很快就印证了 loader 文件传输损坏的想法。我把所有 loader 文件重新复制了一份，故障自然解除。

## 5. yum 安装失败

当安装过程中报告文件缺失或者损坏的时候,并非真的是你的镜像有问题,如图 7-7 所示。检查一下日志输出,多半是因为网络断了。切换到 shell 后用 `route` 和 `ip addr` 去判断。还有一种情况,是把更新的 yum 源和原生的 yum 源这两个内容混放在一起了。由于更新的 yum 源中的软件包不属于原生的 yum 源,导致系统在计算依赖关系时出现了逻辑错误,产生了一个先有鸡还是先有蛋的悖论问题。解决的办法就是把更新部分放到 `%post` 里面执行,或者到网站去更新最新版本的 `repo` 文件。

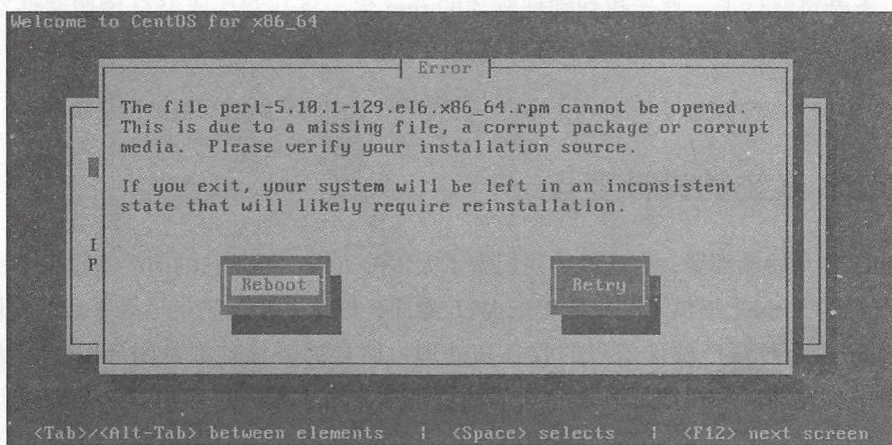


图 7-7 yum 安装失败

## 6. 如何防止恶意 PXE 启动导致原有系统被误装

如果你在维修过程中遇到主板损坏的服务器要特别小心,因为新主板的 BIOS 设置可能启用了 PXE。如果不注意这一点,贸然开机将导致 PXE 引导触发系统重装的悲剧。将 Cobbler 的主配置文件里面的 `pxe_just_once` 选项设置为 1 并重启 `cobbled` 服务即可确保以后生成的 system 只响应一次 PXE 引导请求。对于在此之前已经完成部署的 System,强烈建议使用如下命令禁用 PXE 引导请求。如果 System 需要重装,需要提前手动开启该项功能。

```
cobbler system edit --name=<NAME> --netboot-enabled=False
```

Cobbler 在嵌入 IaaS 系统之后,将由前端 Portal 完成对 Cobbler 的调用。Portal 在调用过程中,要能够根据安装状态随时处理这个开关的启停。在安装过程中有可能因为各种原因导致部署失败,所以在没有完成部署之前, `--netboot-enabled` 的 Value 要始终保持在 True 的状态,给予服务器 Retry 的机会。部署完成后,应当立即禁用 `--netboot-enabled` 选项。

## 7. CentOS 7 对于 Bonding 的影响

2014 年, CentOS 7 正式发布。随着时间的推移,越来越多的生产平台上开始逐步更迭



新一代的操作系统。相对于 CentOS 6，CentOS 7 的对于配置语法上的一些检查更为严格。

以 Bonding 为例，Cobbler 执行安装的时候会将 bond0 的 Type 设置成 Ethernet。这对于 CentOS 6 来讲没有任何问题，但是 CentOS 7 却不允许这种做法，必须要将 Type 修改成 bond 才可以。解决这个问题方法就是在 %post 中执行 sed 替换。

## 8. 如何处理 Linux 内核无法识别新硬件的问题

如果 Linux 内核无法识别新硬件（例如 RAID 卡）有可能导致安装失败。如果采用镜像部署方案就会非常痛苦，因为你需要重新编译内核，还要不断地封包测试。而 Cobbler 只需要一条语句就行了。首先找到硬件驱动的 rpm 包，将其制作成 ISO 镜像文件发布到的 HTTP 上面，然后在 kickstart 里关于分区描述的下面，添加如下一行配置即可。

```
driverdisk --source=http://192.168.0.100/NewDrivers/R920-RAID-H710.iso
```

## 7.4 IaaS 系统的设计要点

Cobbler 作为后端部署，它已经为我们做了太多的工作，但是它的前端却是一个比较薄弱的环节。虽然 Cobbler 提供了一个简单的 Web 界面，但这并不能满足运维平台的实际需求。

Cobbler 缺乏对资产硬件信息的收集和处理，而且部署信息都是闭环的，无法传递到 CMDB 当中。另外，它在地址分配和资源池管理这方面也是一片空白。这个时候，就需要我们自己来“造轮子”。这个轮子是值得造的，我们要构建一个完整的、符合实际需求的 IaaS 系统。

### 7.4.1 交付工作流程定义

在定义需求之前，我们来回顾一下，手动部署一台服务器的操作系统的流程是怎样的。

第一步，完成设备上架和资产录入，资产录入需要自动化部署系统配合。

第二步，设备及配置识别，根据配置设计如何安装操作系统（包括安装系统类型、分区、软件包等）。

第三步，服务器初始化设置，需要对带外管理卡进行设置。

第四步，安装操作系统，这是自动化部署的主要阶段。

第五步，交付系统并撰写实施环境说明，提交各种业务相关的信息。

Cobbler 主要用来完成第四步工作，而剩余工作需要另外的模块负责组织起来。当然，各个模块之间的调用需要通过 Portal 来完成。

设备及配置识别是一个非常重要的环节。在开始部署工作之前，首要任务就是识别服务器的厂商和硬件配置，不同厂商的服务器在初始化配置的过程中都可能存在差异，硬件配置也决定到底如何取安装操作系统。进行识别的关键在于，带外管理卡能够提供足够详细的信息与灵活的查询接口。这也是为什么我们在第 5 章中反复强调 OOB 功能重要性的原



因。如果无法从 OOB 获取足够的信息，则是一个很糟糕的情况，只能等到装载 Preinstall OS 之后，再通过系统命令去获取。OOB 的信息是厂商直接给的，不会有任何错误。而通过系统命令获取，则需要增加很多逻辑判断，有时候会遇到意想不到的情况。比如，我们有一款机型的存储配置是  $4\text{TB} \times 12 + 300\text{GB} \times 2$ ，有个厂商的服务器因为设备布局的问题，2 块小盘和 12 块大盘并不在一个背板上，用的不是同一块 RAID 卡，所以和其他设备相比凭空就多出一块卡，而且这两块 RAID 卡的型号还完全不一样。如果不加判断，想当然地直接用 Megacli 去收集，肯定就会出现问题。

信息录入分成资产、通用和业务三部分。资产信息和通用信息主要依靠带外管理的方式来获取。传统固资盘点的方式是：管理员通过对服务器扫码来生成 Excel 信息表。由于固资的岗位职责和业务没有关系，这个信息表你可以单独保存，但不要直接录入 CMDB 或者 RPDB。当一个真正的 IaaS 系统构建完毕后，至少在运维团队内部，没必要再去做手工盘点。如果服务器入库后还没有加电，从 IaaS 系统的角度上，将它们视作不存在是允许的。此时不要着急，只要你在收货的时候清点好设备数量就行了。信息采集的工作可以等到服务器完成加电之后再谈。毕竟，固资盘点工作和设备入库不同步也是经常发生的。线上的业务信息需要使用 Workflow 来传递参数，所有的业务申请或变更必须经过 Workflow 执行，严禁离线操作。没有经过 Workflow 的环节，意味着信息失联，后续补充就是难以确保准确性。这样一来，除了服务器价格需要手动录入以外，其他信息的来源全都是有据可依的，其准确性和权威性也就不言而喻了。

## 7.4.2 Portal 模块与各组件之间的调用关系

在了解了整个安装部署的工作流程之后，我们就知道应当如何去定义前端 Portal 系统和各个组件之间的调用关系了。我们来看下面这张 IaaS 系统调用关系图，如图 7-8 所示。

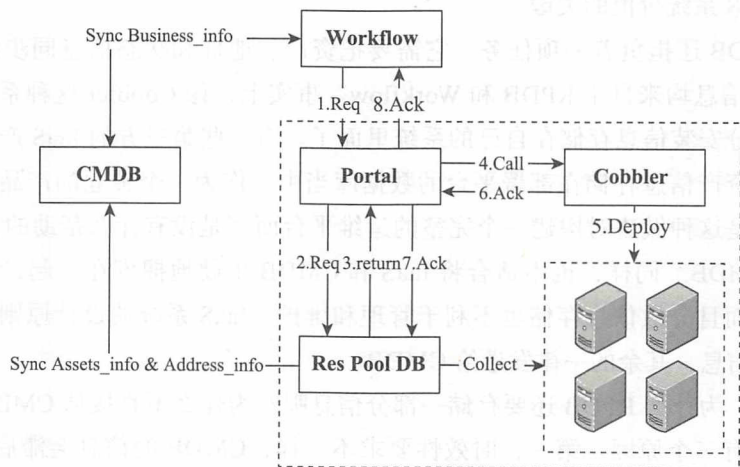


图 7-8 IaaS 系统调用关系图

由虚线所围的这些模块组成了一个标准的 IaaS 系统。其中，Portal 是前端操作页面，它负责响应 Workflow 的请求，并调用后端 Cobbler 执行自动化部署工作。RPDB 是资源池数据库，它负责搜集资产信息和通用信息（IP 地址和服务器状态）。Portal 在请求安装之前，需要到 RPDB 去查询可用资源的情况。

SE 在部署时只关心两件事情：第一，我要部署哪些模板；第二，每个模板需要部署多少台服务器。这些操作是在 Portal 上面完成的，其他的地址分配、设备数量分配等工作都应当由 RPDB 来决定。RPDB 要能够根据硬件配置识别服务器类型，并且实现资源的分配管理，才能告知 SE 能不能执行安装。当然，地址的规划是需要 NE 统一提交的，动态分配由 RPDB 自治。

说到地址自动分配就带来一个问题：RPDB 是如何实现这个功能的呢？因为地址规划和业务是有关系的，只有知道是什么业务才能安排对应的地址段。业务名称是通过 Workflow 在申请时提交的。业务名称如果想做到规范，就需要设置 List 让用户去选择，业务名称的创建就需要由负责业务的管理员来维护，但这个维护有可能因为滞后性会成为部署的瓶颈。更好的办法是让用户选择 VLAN\_ID。VLAN\_ID 代表的是业务类型，因为业务的数量远远大于业务类型的数量，所以通过 VLAN\_ID 可以更好地锁定地址段。由于 Pod 规模的影响，可能一个 VLAN\_ID 对应的不止一个地址段，在不同的数据中心又有不同的 Pod，那么在部署时 IaaS 会选择哪些服务器呢？你可以采取不同的策略来决定如何选择。默认情况下，IaaS 只关心哪些服务器是空闲可用的，并不在乎这些空闲资源的位置。有可能有三个网段都有空闲的资源，如果你想在一次部署过程里尽可能交付比较集中的资源，那么就按照保有量来分配。比如三个网段分别有 10 台、5 台和 8 台，你一次申请了 9 台，那就选择第一个网段。如果你想进行离散分布式的部署，那么就采取每个网段三台的策略。这个策略是可以提供给用户进行选择的。由此看来，RPDB 实际上是 IaaS 动态管理中的一个重点内容，也是决定了 IaaS 系统价值的关键。

另外，RPDB 还担负着一项任务。它需要把资产、地址和状态信息同步给 CMDB，而 CMDB 的所有信息均来自于 RPDB 和 Workflow。事实上，像 Cobbler 这种系统，它在安装时已经把一部分安装信息存储在自己的系统里面了。有一些第三方的 IaaS 产品也采用了类似的方式，把资产信息存储在部署平台的数据库当中。作为一个独立的产品开发，这样做无可厚非。但是这种做法对构建一个完整的运维平台而言是没有什么帮助的，因为 IaaS 系统无法代替 CMDB。同样，也不适合将 IaaS 和 CMDB 生硬地捆绑在一起，那样会导致系统的紧耦合，而且分散信息存储也不利于管理和维护。IaaS 系统的设计原则就是：只存储和部署有关的信息，其余的一律发送给 CMDB。

既然如此，为什么 RPDB 还要存储一部分信息呢？为什么不直接从 CMDB 中提取数据呢？这里主要有三个原因。第一，时效性要求不一样，CMDB 的信息会滞后于 RPDB。第二，内容不一样，CMDB 的信息是和业务、资产紧密相关的，而 RPDB 的信息仅用于部署



安装。第三，对应关系不一样，RPDB 是给 Portal 用的，是 IaaS 系统的一部分，CMDB 是给人用的，属于独立于 IaaS 系统之外的核心数据库。

CMDB 的信息内容理应是最全最新的，但并非所有的信息都会在部署完成后立即成形。这是为什么呢？因为 CMDB 包含了资产、通用和业务三种信息（详情请参见 6.2.4 节的内容），而这三种信息不是同步录入的。资产信息和带外管理地址，在服务器完成加电之后就可以开始采集了。而业务地址则是在系统完成部署后才能够获取，至于业务信息是在流程的不同环节分批次出现的。而且业务信息存在着更新的问题。

例如，你采购了 1000 台服务器。从 IaaS 的角度来说，这 1000 台服务器的操作系统应当一次性完成交付。由于很多信息（像硬件配置、业务部门申请服务器的数量）都是事先定义好的，所以没有必要再等待需求的提交。但对于 PaaS 来说，PE 或者 DBA 可能会先代替业务部门去申请资源，然后再根据各业务部门的实际需要逐步分发。这种场景涉及两次不同的 Workflow，第一次由 PE/DBA 团队向 SE 团队提交资源申请，第二次是业务部门向 PE/DBA 提交申请。对于服务器来说，刚上架时它的状态是 Raw，没有具体的业务信息。此时的 Owner 是归属 IDC 和 SE 的，或者也可以为空，因为此时服务器是裸设备。第一次申请完成后，它的状态变更为 Assigned，有了业务地址，Owner 属于运维团队，第二次申请完成后，它的状态变更为 Online，有了业务名称和 FT\_Rank，Owner 属于业务部门。可以看出，所有信息都是在不断增加和变化的。

所以，你不要指望 CMDB 的信息能够一次到位。而且，CMDB 对资源使用的情况并不敏感。比如，你计划部署 1000 台服务器，但其中有 80 台服务器因为某种原因没能上报到 CMDB 里。对于没有完成信息入库的服务器，我们不建议立即上线使用，可以考虑重装或补装 80 台。也就是说，没有进入 CMDB 的系统不能上线，这样一来，CMDB 的信息是否及时录入不再重要，顶多就是后续同步的问题。但是 RPDB 就不一样了。RPDB 中的信息必须先于安装之前就确定下来，没有资源就不能执行安装。虽然这 80 台服务器在 CMDB 里面还处在 Raw 的状态，对于业务来说就是不存在。但对于 RPDB 而言，不论是地址还是服务器，这 80 个资源就是没有了。

CMDB 是一个相对静态的信息系统，它反映的是一个状态。在从状态 A 行进到状态 B 的这个过程当中，只要没有走到状态 B，那么它就应该一直处于 A 的状态。这个状态值的修改，是要依靠触发事件或者流程变更来驱动的。而 RPDB 则是动态实时的。如果一个 IP 地址资源被分配出去了，它立即就会减少一个地址，即便这台服务器没有安装成功也一样。少了就是少了，它不关心结果，只看实时的变化。

## 7.5 制作 KVM 虚拟机模板

KVM 已经是目前生产系统中普遍使用的一种技术了。KVM 在部署过程中，还是有很



多和物理服务器不太相同的地方，需要我们格外注意。

### 7.5.1 虚拟机网络环境部署

#### 1. 关于桥接

说到桥接，可能有些同学没有搞清楚 Linux 里面 Bridge 设备和网络之间的关系，因此在桥接的配置上总是会出现各种莫名其妙的错误。关于这个问题请大家参见图 7-9，如果把宿主机看作一个独立的网络环境，那么作为桥接的 brX 设备相当于一台虚拟交换机。这台交换机的 Uplink 就是宿主机的物理网卡。如果宿主机上的网络设备接口（例如 bondX、emX、ethX、pXpY 等）被桥接到 brX 上面，那么它们则会成为这台虚拟机交换机上面的 Port。这些 Port 属于二层，是不能配置 IP 的，所以在桥接之后原有的 IP 地址需要迁移到 brX 上，用来实现对于宿主机的访问。此时，物理接口处在混杂模式下，它和 brX 的 MAC 地址是一样的。从上层通信的角度上看只能发现 brX，brX 下面的细节对于上层来说是不可见的。brX 设备上是否配置 IP 地址并不影响虚拟机的通信。如果 brX 上没有配置 IP 地址，或者 IP 地址还依旧落在已经桥接到 brX 的物理接口上面，那么在重启后，宿主机将没有 IP 地址，路由表中只有回环地址的路由条目。但是虚拟机依旧能和外界通信，此时的 brX 相当于一根导线，把虚拟机的虚拟网卡和物理交换机的 Port 直接对接了起来。

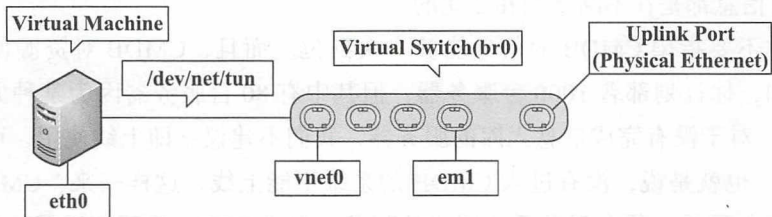


图 7-9 桥接设备与虚拟机之间的关系

#### 2. 配置桥接设备

弄懂了图 7-9，我们就知道了，为什么桥接要这样去配置。一般情况下，没有 VLAN 的桥接设置的步骤就是：把物理接口桥接到 brX 设备上，再把 IP 地址设置在桥接设备上。如果涉及 VLAN 设备，就需要三个步骤。基于原来的物理接口再创建一个带有 VLAN TAG 的物理接口，把这个带有 VLAN TAG 的物理接口桥接到 brX 上面，而宿主机 IP 地址的配置有两种情况。第一种情况是直接把宿主机的 IP 地址配置在 brX 上面，此时这个地址也是带 VLAN TAG 的，如果宿主机的 IP 地址段在原来规划时是不带 VLAN 的，那肯定不通。这就需要转向第二种情况，让宿主机的 IP 地址保留在原有的物理接口上不变，此时这个地址还是 Native 的，我们要在交换机上做一条配置，允许 Native VLAN 可以访问那个业务所在的 VLAN。我们在前面章节中讲过，VLAN 是用于前端业务的，VLAN 很可能是多个。那么，一个宿主机上就需要有多个 VLAN，而宿主机本身是不涉及业务访问的，我们可以

使用第二种方法，将宿主机的 IP 地址放置到 Native VLAN 中，在 ACL 列表中允许 Native VLAN 访问所有 VLAN 设备即可。

```
// 没有 VLAN 的桥接设置
/etc/sysconfig/network-scripts/ifcfg-bond0
    DEVICE=bond0
    ONBOOT=yes
    BONDING_OPTS="miimon=100 mode=4 xmit_hash_policy=layer2+3"
    TYPE=Ethernet
    BOOTPROTO=none
    BRIDGE=br0

/etc/sysconfig/network-scripts/ifcfg-br0
    DEVICE=br0
    TYPE=BRIDGE
    BOOTPROTO=none
    ONBOOT=yes
    IPADDR=192.168.1.100
    NETMASK=255.255.255.0

// 带有 VLAN 的桥接设置
/etc/sysconfig/network-scripts/ifcfg-bond0
    DEVICE=bond0
    TYPE=Ethernet
    BOOTPROTO=none
    ONBOOT=yes
    BONDING_OPTS="miimon=100 mode=4 xmit_hash_policy=layer2+3"
    IPADDR=192.168.1.100
    NETMASK=255.255.255.0

/etc/sysconfig/network-scripts/ifcfg-bond0.200
    DEVICE=bond0.200
    TYPE=Ethernet
    BOOTPROTO=none
    ONBOOT=yes
    VLAN=yes
    ONPARENT=yes
    BRIDGE=br200

/etc/sysconfig/network-scripts/ifcfg-br200
    DEVICE=br200
    TYPE=BRIDGE
    BOOTPROTO=none
    ONBOOT=yes
    BOOTPROTO=static
```

### 3. 虚拟机 default 网络删除

libvirt 安装完毕后，会自动生成一个 192.168.122.0/24 的默认网络。但我们并不需要它，可通过 virsh 自带的 net-destroy 命令将其删除。该定义网络的配置文件位于来自 /var/



lib/libvirt/network/default.xml。执行 net-destroy 命令，实际上就是删除了这个文件。

default.xml 的详细内容如下所示。

```
[root@station103 ~]# cat /var/lib/libvirt/network/default.xml
<!--
WARNING: THIS IS AN AUTO-GENERATED FILE. CHANGES TO IT ARE LIKELY TO BE
OVERWRITTEN AND LOST. Changes to this xml configuration should be made using:
    virsh net-edit default
or other application using the libvirt API.
-->

<network>
  <name>default</name>
  <uuid>ald0fb1f-f353-4e87-b253-0130a63aea4d</uuid>
  <forward mode='nat' />
  <bridge name='virbr0' stp='on' delay='0' />
  <mac address='52:54:00:09:45:11' />
  <ip address='192.168.122.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.122.2' end='192.168.122.254' />
    </dhcp>
  </ip>
</network>
```

## 7.5.2 创建虚拟机镜像模板

### 1. 使用 Cobbler 安装虚拟机

通过 Cobbler 安装虚拟机是最简单有效的方式。它的基本步骤和物理服务器的部署无异，虚拟机的 PXE 启动可以借助 gPXE 来完成，gPXE 的具体操作请参见如下链接：[http://cobbler.github.io/manuals/2.6.0/4/13\\_-\\_Using\\_gPXE.html](http://cobbler.github.io/manuals/2.6.0/4/13_-_Using_gPXE.html)。

### 2. VNC 模式安装虚拟机

手动安装虚拟机的好处是可以随时调整系统状态。生产系统通常不会安装图形库，没有办法在宿主机上面安装 Virt-Manager 这种图形化工具。而且生产环境的访问会受到堡垒主机的限制，我们也不打算在它上面去解决 X11 Forwarding 的问题。我们可以使用 VNC 的方式来实现图形化安装。

首先，执行如下命令创建一个镜像文件。

```
# qemu-img create -f qcow2 /export/kvm/CentOS-6-u5.qcow2.img 10G
```

然后，执行如下命令开始安装虚拟机。

```
# virt-install -n CentOS-6-u5 \
  --vcpus 1 -r 512 \
  --disk path=/export/kvm/CentOS-6-u5.qcow2.img,format=qcow2,size=10 \
  --cdrom=/export/kvm/CentOS-6.5-x86_64-bin-DVD1.iso \
  --network bridge=br0 \
```





```
--os-type=linux --os-variant=rhel6 --noautoconsole \
--vnc --vnclisten=0.0.0.0 --vncport=6000
```

上述命令可以创建一台磁盘空间为 10GB 的虚拟机，硬件配置参数可以根据实际需求进行更改。virt-install 命令执行后，使用 VNC Viewer 客户端连接宿主机地址 x.x.x.x:6000，即可开启终端控制台的界面。

--vncport=5901 和 --vncport=1 是等价的，建议大家还是使用 5901 这种方式显式指定 VNC 端口。如果不想指定端口号，可以使用 --vncport=-1，让 libvirtd 自行分配，但这样做需要在 virt-install 命令执行后通过 ps 命令自己查询 VNC 的端口号。

如果你在使用 VNC Viewer 连接终端时，出现 VNC Viewer 窗口闪了一下瞬间又消失了，这是因为 VNC Viewer 的颜色模式不兼容所造成的。解决方式如下：选择 Options->Advanced->Expert 中的 ColorLevel，将默认值 256 色模式（pal8）修改成全彩模式（full）即可。

### 3. 文本模式安装虚拟机

如果连 VNC Viewer 的使用条件都不具备，那么还有一招：使用文本模式完成虚拟机的安装。这需要在 virt-install 命令后面再多增加一个 --extra-args 的参数。

```
--extra-args='console=ttyS0 console=ttyS0,115200n8'
```

## 7.5.3 虚拟机克隆

### 1. virt-clone

如果虚拟机完全通过 Cobbler 来安装，由于网络带宽的限制，速度上会受到一定影响。创建一个虚拟机模板，然后在宿主机本地执行 virt-clone 来实现批量克隆是一种更好的解决方案。

```
# virt-clone -o <SOURCE_NAME> -n <NEW_NAME> -f <NEW_IMAGE>
```

### 2. virt-sysprep

刚刚克隆完成的虚拟机是不能直接使用的。克隆虚拟机的配置文件来自于模板，除了 MAC 地址以外，大部分内容还都是一致的（例如主机名、IP 地址、UUID、ssh-key、udev-persistent-net 名称等）。这些相同的内容，将会在虚拟机之间的网络通信中产生冲突，必须擦掉重写。这就用到了 virt-sysprep 命令。

virt-sysprep 既可以用在 virt-clone 之前，也可以用在 virt-clone 之后。如果 virt-sysprep 在 virt-clone 之前执行，主要作用是配置信息的擦除。而 virt-sysprep 在 virt-clone 之后执行，通常是用来生成一个新的虚拟机实例。通过叠加 --firstboot 选项，可以执行自定义脚本，用于修改主机名、IP 地址等内容，在配置上将获得最大的灵活性。

```
# virt-sysprep -d <NEW_NAME> -h <HOSTNAME> --firstboot <SCRIPT>
```

### 3. guestfish 和 virt-edit

如果你想编辑一个镜像文件，但又不想开启虚拟机，那么可以使用 guestfish。guestfish 是用来安全地探索一个未知镜像的最好方式。一个有经验的 SE，在不了解镜像文件的前提下，是绝不会贸然开启虚拟机的。未知的镜像文件就像潘多拉魔盒，一旦开启有可能会出现意想不到“惊喜”，例如镜像的 IP 地址和线上生产的地址是冲突的。

因为 guestfish 需要挂载，所以镜像文件越大，挂载的时间就越长。virt-edit 也是一款和 guestfish 类似的工具。但 virt-edit 每次只能打开一个文件，不适合修改多个文件，而且用它修改文件的前提是必须清楚地了解镜像文件中都有哪些文件。

不论是使用 guestfish 还是 virt-edit，在修改文件的时候都要注意一点，修改对象必须是源文件而非符号链接。此时，编辑对象是被当作一个普通文件来处理的。你如果修改的是一个符号链接，实际文件则不会有任何变化，而且符号链接会被破坏，变成一个普通文件。/etc/ 目录下的 GRUB 和 SELinux 的配置文件都是属于这类例子。

```
# guestfish -i -a <IMAGE>
# virt-edit -a <IMAGE> -f <FILE>
```

## 7.5.4 虚拟机设备调整

### 1. 设置 SN

上一章我们在定义 CMDB 表结构的时候，有一项是 Product\_SN，代表设备的 SN 号。物理服务器的 SN 信息是烧录在 BIOS 中的。虚拟机也可以通过定义 XML 配置文件来设定 SN 信息。但是和物理服务器不同，这个 SN 信息的定义不是随机的，而是要和其所在的宿主机关联起来。我们使用 SN-N 的方式来描述虚拟机的 SN 信息。这里的 SN 指的是宿主机的 SN 信息，而 N 则代表虚拟机的序号。例如 33CQW22-1，代表的就是 SN 信息为 33CQW22 的宿主机上的第一台虚拟机。这样的安排是有特别考虑的。如果虚拟机远程访问失败，在进行故障处理时，SE 必须要登录到宿主机上面实施维护工作。因此，SE 必须要了解故障虚拟机隶属于哪一台宿主机。而使用这种关联方式的 SN 信息是最简单有效的解决方案。

下面就是一个定义虚拟机 SN 信息的 XML 配置文件示例。

```
<sysinfo type='smbios'>
  <system>
    <entry name='manufacturer'>Nobody</entry>
    <entry name='product'>Nobody-KVM-VM</entry>
    <entry name='serial'>33CQW22-1</entry>
  </system>
</sysinfo>
```

### 2. 调整 CPU、内存

```
# virsh setvcpus <DOMAIN> <COUNT> [--maximum] --current
```



```
# virsh setvcpus <DOMAIN> <COUNT> [--maximum] --config
# virsh setmem <DOMAIN> <SIZE> --current
# virsh setmem <DOMAIN> <SIZE> --config
# virsh setmaxmem <DOMAIN> <SIZE> --current
# virsh setmaxmem <DOMAIN> <SIZE> --config
```

--current 只会影响当前虚拟机的状态，虚拟机重启后设置失效。如果需要这个配置永久生效，则需要改成 --config。指定 --config 意味着这项变更将修改 XML 配置文件中的内容。

### 3. 增加 / 删除网卡

```
# virsh attach-interface <DOMAIN> --type bridge --source br0 \
    [--model virtio] [--config]
# virsh detach-interface <DOMAIN> --type bridge --mac <VM_MAC> [--config]
```

这里的斜体字 br0 是一个示例名称，实际使用时请根据你的实际情况进行替换。virtio 模式可以让网卡进一步提升性能，采用默认模式时，其设备是工作在模拟 RTL 8139 芯片的 100M 全双工模式下。

### 4. 增加 / 删除磁盘

```
# virsh attach-disk <DOMAIN> --source <NEW_IMAGE> --target vdb --config
# virsh attach-disk <DOMAIN> --source <NEW_IMAGE> --target hdc \
    --type=cdrom --mode=readonly --config
# virsh detach-disk <DOMAIN> --target <DEVICE> --config
```

### 5. 更换虚拟机光盘

执行如下命令，检查光盘设备名称。

```
# virsh domblklist winxp
Target      Source
-----
hda          /export/kvm/winxp.img
hdc          /export/iso/WinXP_SP3.iso
```

使用 change-media 完成更换光盘的操作。

```
# virsh change-media --domain winxp hdc \
    /export/iso/virtio-win-0.1.iso
```

更换完成后，可再次使用 domblklist 检查执行结果。

```
# virsh domblklist winxp
Target      Source
-----
hda          /export/kvm/winxp.img
hdc          /export/iso/virtio-win-0.1.iso
```

## 7.5.5 VPC 的支持

虚拟私有化云（Virtual Private Cloud, VPC）是近年来比较流行的一种新型的弹性云构





建模式。它有些类似于 VPN 的概念。通过 VPC 技术，用户可以将那些原本相互隔离的云，相互连接成一个整体，好像它们原本就同属一个云一样。而且用户可以完全自定义所有的网络配置和安全策略，构建属于自己的私有化云平台。

从操作系统的角度讲，由于虚拟化是构建在宿主机上的，要实现用户自定义安全策略，比较直接的解决方案就是采用 iptables。

### 1. iptables 策略设置

我们都知道 iptables 的规则匹配是自顶而下的，如果遇到条件匹配且有明确 Target (RETURN 或 LOG 除外) 的规则，iptables 将不再继续向下匹配；如果没有找到任何匹配的规则，则遵循 chain 的默认规则。由于虚拟机是随机分配给用户的，而且每一个用户都可以修改安全策略，如果所有用户的规则都位于同一个 Chain 里，那么在用户自定义策略的过程中就会彻底乱套。

假设规则的添加方式是 Append，那么最先提交的用户将会获得极大的话语权。比如，用户 A 租赁了一台 IP 地址为 192.168.1.1 的虚拟主机，并且他率先提交了安全策略——只允许外网登录 192.168.1.1 的 ssh 和 http 服务，其余的全部拒绝。那么，这将导致位于该策略后面的所有规则全部失效，等于用户 A 独霸了整台宿主机。

为了解决这个问题，必须将用户的流量进行分流。不能因为用户 A 修改策略而影响到用户 B 的正常使用。由于每个用户使用了不同的虚拟网卡，所以拆分流量需要在二层上面完成。我们设计 iptables 架构的主旨就在于把不同用户的流量拆分开来。

下面是一个 iptables 安全策略架构的示例。VPC-FW-FORWARD 是一个自定义的 Chain。所有有关于 VPC 的策略都将放置到这个 Chain 中。首先，我们将流量按照进出方向分别拆成 neutron-sg-in 和 neutron-sg-out。然后，按照虚拟机的网卡把不同用户的流量拆分开来。每一个虚拟机下面又分为默认和自定义两个部分：default 用来放置我们预先定义的一些基本规则，用户所做的全部修改都将在 custom 里面完成添加。这样我们就把用户策略锁定在了 custom 里面，而宿主机的安全策略架构则不会受到用户自定义的影响。

```
/ VPC-FW-FORWARD
|—— neutron-sg-in
|   |—— neutron-sg-in-vm001
|   |   |—— vm001-out-custom
|   |   |—— vm001-in-default
|   |—— neutron-sg-in-vm002
|   |   |—— vm002-in-custom
|   |   |—— vm002-in-default
|   ...
|   |—— neutron-sg-in-vm00N
|   |   |—— vm00N-in-custom
|   |   |—— vm00N-in-default
|—— neutron-sg-out
```

```

|   |—— neutron-sg-out-vm001
|   |   |—— vm001-out-custom
|   |   |—— vm001-out-default
|   |—— neutron-sg-out-vm002
|   |   |—— vm002-out-custom
|   |   |—— vm002-out-default
|   ...
|   |—— neutron-sg-out-vm00N
|   |   |—— vm00N-in-custom
|   |   |—— vm00N-in-default

```

下面是一组实现上述安全策略架构的 iptables 示例命令，供读者在实际应用测试时进行参考。

```

# Create custom chains.
iptables -N VPC-FW-FORWARD
iptables -N neutron-sg-in
iptables -N neutron-sg-out
iptables -N neutron-sg-in-vm001
iptables -N neutron-sg-out-vm001
iptables -N vm001-out-custom
iptables -N vm001-in-custom
iptables -N vm001-in-default
iptables -N vm001-out-default

# Create a virtual machine default chains.
iptables -A vm001-in-default -p icmp -j ACCEPT
iptables -A vm001-in-default --dport 22 -j ACCEPT
iptables -A vm001-in-default -j DROP
iptables -A vm001-out-default -p icmp -j ACCEPT
iptables -A vm001-out-default --dport 80 -j ACCEPT
iptables -A vm001-out-default -j DROP

# Create a virtual machine chains.
iptables -A neutron-sg-in-vm001 -j vm001-in-custom
iptables -A neutron-sg-in-vm001 -j vm001-in-default
iptables -A neutron-sg-in-vm001 -j RETURN
iptables -A neutron-sg-out-vm001 -j vm001-out-custom
iptables -A neutron-sg-out-vm001 -j vm001-out-default
iptables -A neutron-sg-out-vm001 -j RETURN

# Create other chains.
iptables -A neutron-sg-in -m physdev --physdev-in vnet1 --physdev-is-bridged -j
    neutron-sg-in-vm001
iptables -A neutron-sg-out -m physdev --physdev-out vnet1 --physdev-is-bridged
    -j neutron-sg-out-vm001
iptables -A VPC-FW-FORWARD -m physdev --physdev-in vnet1 --physdev-is-bridged -j
    neutron-sg-in
iptables -A VPC-FW-FORWARD -m physdev --physdev-out vnet1 --physdev-is-bridged
    -j neutron-sg-out

```



```
iptables -A FORWARD -j VPC-FW-FORWARD
```

## 2. iptables 策略设置

在上述的示例命令中，我们是通过 `physdev` 信息来实现流量拆分。要让 `iptables` 识别 Bridge 设备上的流量，必须开启如下内核参数，这些内核参数是基于 `bridge` 模块加载后才有的。

```
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-arptables = 1
bridge-nf-filter-vlan-tagged = 1
```

## 3. 指定网络设备名称

默认情况下，KVM 的网络设备名称是 `vnetX`，`X` 的数值是根据启动顺序依次分配的，因为启动顺序是随机的，所以网络设备名称也是随机的。

那么，在设置安全策略的时候就会出现一个问题。我们是通过 `physdev` 信息来实现流量拆分，也就是这里面的 `vnetX`。如果 `vnetX` 这个设备名称不能固定，在虚拟机重启后，原本属于用户 A 的安全策略有可能会被应用到用户 B 的虚拟主机上面。轻则导致策略失效，重则引发所有用户都无法访问自己的虚拟机主机。所以，必须锁定 `vnetX` 这个设备名称。

要指定 `vnet`，不能使用 `<target dev='vnetX'/>` 这种范式，这种命名方法是错误的。`vnet` 和 `virbr` 是 `libvirt` 的保留关键字，当使用这两个字段作为名称前缀时，`libvirt` 会直接无视。因此，必须要改成其他名字。笔者测试过 `vpcX` 这种形式是可以的，但建议读者最好不要以 `v` 字母开头。因为，笔者第一次测试时，采用了 `vn` 开头的名称发现同样无法生效，笔者猜想有可能在匹配过程中存在贪婪模式而造成的。如果你非要用 `v` 开头的字母，也可以尝试采用这种方法，先修改成一个中间的临时名称并使其生效，通知 `libvirt` 状态先发生一次变更确认，然后再修改回你所希望的名称。

使之生效的方法是重启 `libvirtd` 服务，并重启虚拟机，最后可以使用命令 `virsh domiflist <DOMAIN>` 来检查修改后的效果。

```
# virsh domiflist redhat
Interface Type      Source      Model      MAC
-----
vpc11      bridge      br0         virtio      52:54:00:39:fc:a5
```

关于 `vnet` 设备的关键字声明在原文记载如下。

If no target is specified, certain hypervisors will automatically generate a name for the created tun device. This name can be manually specified, however the name should not start with either 'vnet' or 'vif', which are prefixes reserved by libvirt and certain hypervisors. Manually specified targets using these prefixes may be ignored.

以上来自 <http://libvirt.org/formatdomain.html#elementsNICSTargetOverride>



## 7.6 本章小结

本章讨论了如何构建一个 IaaS 平台系统。针对自动化部署方案的选择,在业界有两种不同的观点。笔者建议选用 Cobbler+Portal+RPDB 的方式来构建一套完整的 IaaS 平台系统。本章详细地讨论了 Cobbler 的整体架构,并给出实例化配置方案和常见故障的处理方法。同时,对于 Portal 和 RPDB 的构建,也提供了相应的设计思路与要点提示。最后,本章分享了 KVM 虚拟机在部署实践中的一些典型案例。

## 第 8 章

# 构建域名解析服务

当你和一个资深系统管理员讨论互联网技术时，他首先想到的就会是域名解析服务，也就是我们常说的 DNS（Domain Name System）。域名和 IP 地址都是互联网上的唯一身份标识，但是相对于难记的 IP 地址，基于域名访问的寻址方式仍旧是绝大多数互联网应用的首选。

说起 DNS，不由得让我想起了很久以前的某个人。他从入职的那天起，就被领导分配负责 DNS 系统的建设和维护工作。这厮扑在这个项目上一干就是两年多，除了 DNS 和领导交办的事务以外，组内的其他工作很少参与。他平时把很多时间都花在各种技术研究上，整个系统组的人全都忙得焦头烂额的时候，他身为 SE 却连机房都没怎么进过，工作效率低不说，还经常和其他团队的人发生冲突。他把 DNS 系统做得异常复杂，而且还没有文档输出，弄得谁也接管不了，“成功地”建立起了核心系统维护工作的单点隐患。有一次，他的一个同事要升职，结果他就跑去老板那里去闹，并用离职的手段相威胁。最后，老板因为害怕这么重要的 DNS 系统出问题，就把升职加薪的机会留给他了。

什么，你想问他是谁啊？呵呵，这位同学，我发现你的关注焦点还真是有点儿特别啊。那什么，咳咳，我的意思是说：第一，DNS 很重要，它是所有网络访问的基础服务；第二，DNS 很深奥，如果你深入研究下去可以挖掘出很多有价值的内容。嗯，就是这些，你明白了吗？

## 8.1 写在前面的话

本章首先对传统 DNS 一些关键的知识要点进行讲解，并且指出 DNS 系统构建时的一些注意事项，然后为读者介绍当前主流 DNS 技术的实际应用和关键配置过程。

为了让读者理解起来更加直观，笔者在一台废弃的服务器上构建了一个测试环境。BIND 是 Linux 系统上使用最多的 DNS 软件，笔者本次所采用的是 CentOS 6.5 原生的 bind-9.8.2-0.17.rc1.el6\_4.6。

BIND (Berkeley Internet Name Domain) 是美国加州大学 Berkeley 分校开发和维护的一款开源的 DNS 软件,也是目前世界上使用最为广泛的 DNS 软件,支持各种平台。由于 BIND 在 8.X、9.1 以及 9.1 以后的各个版本之间存在着很多差异,因此,本文中所涉及的一些功能有可能在老版本的 BIND 中是无法支持的。如果没有特别说明,本章中提到的 BIND 都是指 bind-9.8.2-0.17.rc1.el6\_4.6 这个版本。本章中使用大写的 BIND 时,是指由 BIND 管理的 DNS 服务。本章中使用小写的 bind 时,是指 bind-9.8.2-0.17.rc1.el6\_4.6 这个软件包。

## 8.2 首先做好一个传统的 DNS 管理员

本来开篇我应当直截了当地介绍,我们是如何实现高大上的 Anycast DNS 和 HTTP DNS 的。因为我从没打算照着 BIND 手册,连篇累牍地给大家介绍什么是 DNS,以及怎么配置一个 DNS 服务器这些网上都能搜到的内容。但是等到真正动笔的时候,我又有些犹豫了,是不是应该在讲高科技之前,先把一些基础问题消化掉呢?

事实上,你去网络上搜索一篇配置文档可以非常轻松地搭建起一个 DNS 服务。然而,这不代表你真正地理解了 DNS。这就和英语不好但不妨碍唱英文歌一样。于是,我决定把基础问题,也就是那些经常被人问来问去的“十万个为什么”的东西解释清楚,再开始深入讨论新技术。

首先,我们要把 DNS 服务器的角色名称搞清楚。现在在国内,很多人都把主从角色分别称作 Master 和 Slave,可是很多国外的文献或者书籍上却出现了 Primary 的说法。原来早先 DNS 规范将主从角色定义为 Primary Master 和 Secondary Master。如果你搞过 Windows Active Directory,应该对这两个名字不陌生。很多上年纪的系统管理员在提及主从关系时,依旧会保留这种正统的称谓。不过后来,这种文化没有流传下来,慢慢就被大家给叫乱了。Secondary Master 被人们称作 Slave,而 Primary Master 被称作 Primary。而相对于 Slave,很多人又把 Primary Master 叫作 Master。一时间名字满天飞。反正我个人觉得 Primary 和 Slave 看上去有点儿不搭调。为了便于理解,我们在这里统一一下名称,在后续的论述当中,我们将 Primary Master 称为 Master Server,将 Second Master 称为 Slave Server。

好,接下来进入科普时间。我们假设读者都是有一定 DNS 基础和经验的管理人员,本章对于那些最基础的内容不再浪费篇幅赘述,我只挑选一些 DNS 当中容易混淆、难于理解的关键性核心问题一一进行拆解,为后面的技术讲解做好铺垫工作。

### 1. SN 是如何工作的

SN 是区域 (zone) 文件的版本修订号,在主从同步时,named 会通过比较 SN 的大小来判断区域文件是否需要更新。SN 是一个 32 位的无符号整数,它的数字空间范围在  $0 \sim 2^{32}-1$  ( $0 \sim 4\,294\,967\,295$ ) 之间。

那么,如何比较两个 SN 之间的大小呢?根据 RFC 1035 的描述,SN 采用的计算方法



是 SSA (Sequence Space Arithmetic)。

SSA 有点类似于“石头剪刀布”的游戏，它规定在  $2^n$  的空间里，数字是首尾相接，呈环状排列的。在 SSA 的空间里，任意一个数字，向后增加，都有  $2^{n/2}-1$  个数字比它大，向前减少，都有  $2^{n/2}-1$  个数字比它小。这么说有点儿乱，我们来举一个例子，大家就清楚了。

假设  $n=2$ ，在这个 SSA 的空间里就有 0、1、2、3 四个数字，它们的排列如图 8-1 所示。根据 SSA 的法则， $3>2$ ， $2>1$ ， $1>0$ ， $0>3$ 。其中 0 和 2，1 和 3 这两对数字之间是没有办法相比较的。因为，如果假设  $2>0$ ， $3>1$ ，而前面已经说了  $1>0$ ， $0>3$ ，那么按照这个逻辑就会推导出  $1>3$ ，正好和一开始的假设是相悖的。所以我们说 0 和 2，1 和 3 之间是没有办法相比较的。也就是说，任何一个数字和它对面的数字都是没有办法相比较的。

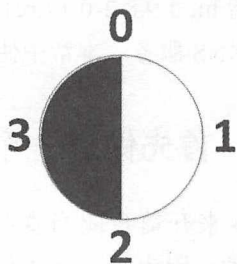


图 8-1 当  $n=2$  时 SSA 的关系描述

按照 SSA 的规定，当  $n=2$  时，任何一个数字都有 1 个数字 ( $2^{2/2}-1=1$ ) 比它大，也都有 1 个数字 ( $2^{2/2}-1=1$ ) 比它小。按照图 8-1 所示，拿 0 来举例，0 的对面是 2，我们按照顺时针转动，从 0 到达 2 (也就是  $2^{2/2}=2$ ) 之前的数字都比 0 大，而剩下的都比 0 小。这么看，有点儿像太极图对不对？我们发现，0 和它对面的数字 2 把整个圆分成了两半，白色那一半里面的数字都是大于 0 的，黑色那一半里面的数字都是小于 0 的。

了解了 SN 数值的大小是如何比较的，我们接下来说一说 SN 数值是怎么设置的。通常管理员习惯采取 YYYYMMDDNN 这种日期 + 数字的方式。但是这种方式限定我们每天对这个区域文件最多只能修改 100 次。因为前面的日期是不能变更的，后面的两位数字只能表达 0~99。对于一般场景来说，这也不是什么问题。区域文件能够修改 100 次也算是很大的工作量了。不过，区域文件如果修改次数非常频繁 (例如启用 BIND 9 的动态更新)，就必须放弃这种方案，采用递增加 1 的形式来记录版本修订号。这种做法也有麻烦，就是无法像前一种方案那样，直观地知晓某一段单位时间内我们修改的记录次数。假设说，我想每个月统计一次，还得把上一次的记录翻出来，两个数字相减才能得到答案。如果每月初，我都能够把数字归零，重新统计该多好。但是 SN 的数值是非常大的，正常情况下想要归零，估计等你退休都不见得看到那一天。然而，我们是有办法的。以刚才讲过的内容为基础，我继续为大家说明，在这种情况下该如何去做。

还是举例说明，SN 是一个 32 位的无符号整数，如果在 SSA 算法当中，应当是  $n=32$ 。但是，我可不打算把大家和我自己搞晕。我会把  $n$  值设置得小一点儿，读者朋友们能够理解就好了。

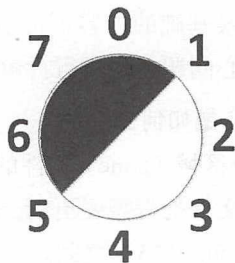


图 8-2 当  $n=3$  时 SSA 的关系描述之 SN=1

是1, 为了能快速回到0, 总共需要三个步骤。根据前面我们学到的方法可知, 比1大的是2、3、4。首先, 把SN从1直接增加到4, 如图8-3所示。接下来, 比4大的是5、6、7, 再把SN从4直接增加到7, 如图8-4所示。此时, 7已经到达了这一个轮回的尽头了。当一个数字在增加的过程中大于 $2^n$ 时, 就要减掉 $2^n$ 。比如本例中,  $7+3=10$ ,  $10>2^3$ ,  $10-2^3=10-8=2$ 。利用这个方法, 我们在第三步就可以让SN重新归零了。

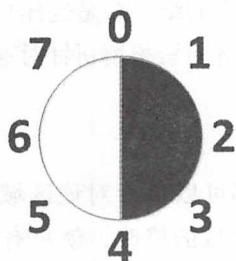


图 8-3 当  $n=3$  时 SSA 的关系描述之  $SN=4$

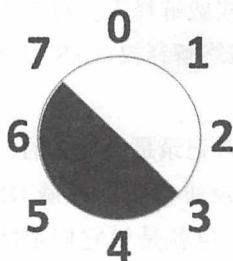


图 8-4 当  $n=3$  时 SSA 的关系描述之  $SN=7$

如果想将SN重新归零, 可以采用这种大步行进的方式, 快速地穿越并重新回到初世代。最大一步可以迈出 $2^{n/2}-1$ 那么远, 而我们这里的 $SN=1$ 是一种特殊情况, 没有人一开始就要把SN归零的, 因此实际上只需要两步就可以完成归零的任务了。

## 2. BIND 如何配置

我已经好久没有配置BIND了, 要不要到网上去搜一下? 完全没有那个必要。还有什么文档会比bind自身的示例文件更权威、更准确呢? bind安装完毕后, 在`/usr/share/doc/bind-9.8.2/sample/`目录下就有现成的示例文件供我们参考。

`./etc/named.conf` 是主配置文件的示例。  
`./etc/named.rfc1912.zones` 是 zone 文件的示例。  
`./var/named/named.empty` 是资源记录语法的示例。

## 3. 子域委派

如果一家企业旗下拥有多个分支机构, 父域的管理员可能没有太多精力去担负所有域名解析的维护工作。此时就需要把子域授权给其他人员来维护, 这种权利下放称作委派。子域委派需要父域的管理员完成两项工作: 创建NS记录指定子域要授权给哪一台DNS Server, 并提供该DNS Server的A记录解析。

例如, 我们有一个域叫`example.com`, 它的权威DNS Server是`station103`, 现在要把子域`phi.example.com`委派给`station203`来管理。那么只需要在`station103`上`example.com`的zone文件中进行如下配置即可。

```
phi          IN  NS  station203.example.com.
station203   IN  A   10.225.11.203
```

#### 4. IN 是什么

上面的示例中，IN 是资源记录中的类型，代表 Internet。通常，它不该出现在内网的域名解析当中，只有公网域名解析才需要使用类型 IN。

#### 5. NS 记录是做什么的

子域委派里，我们提到了 NS 记录。NS (Name Server) 记录代表谁才是这个区域 (zone) 的权威解释人。日常生活中，经常会看到一些促销广告的结尾总是这样写着：厂商对此拥有最终解释权。NS 记录中所指明的那台 DNS Server，才有资格解析针对这个区域的查询请求。

#### 6. SOA 记录是做什么的

NS 记录所指定的权威 DNS Server 允许有多个，它们都可以负责对该区域进行权威域名解析，也就是说它们都能读取这个区域。但是，对于区域的修改，就只有一台 DNS Server 能做到。SOA (Start of Authority) 记录就是表明谁才是这个区域的老大。SOA 有且只有一个，只有 SOA 所指定的 DNS Server 才能修改这个区域。

那么，Master Server 和 SOA 之间又有什么关系呢？没有绝对关系。SOA 所指的 DNS Server 只代表它是该区域的负责人，对于这个区域来说它就是 Master Server 的角色，但不见得它在所有其他区域中都是这样。虽然通常情况下，我们的 DNS Server 往往只承担一种角色。但一定要在 SOA 和 Master Server 之间建立联系，就有点儿过于牵强了。

#### 7. 还要不要设置 root hint 和 loopback

从 BIND 9 开始已经内置了 root hint。至于 loopback，除非你的应用程序需要 127.0.0.0/8 这个段的解析，否则大可不必。当然如果你强迫症犯了，非要写上，也没有问题。

#### 8. zone 文件中的三个 TTL

TTL 是 Time To Live 的缩写，这里的 TTL 是指一条数据在缓存里可以停留的时间。每一个 zone 文件都有三个位置可以写入 TTL。

第一个是资源记录当中的 TTL，它是单独为某一条资源记录而设定的 TTL。

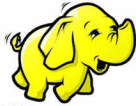
| {NAME} | {TTL} | {CLASS} | {TYPE} | {RDATA}  |
|--------|-------|---------|--------|----------|
| www    | 60    | IN      | A      | 11.0.0.1 |

第二个是独占一行的 TTL，它通常会被写在 zone 文件的行首，它代表为一组资源记录设定的 TTL。凡是位于它下方那些没有特别指定 TTL 的资源记录，都将使用这个 TTL 设置的值。这个 TTL 也可以写多个，但通常只写一个，所以它有时被人误认为是全局 TTL。

第三个是位于 SOA 记录中的 TTL，在 bind 8.2 及以后的版本中，这个 TTL 指的是 Negative Response TTL (RFC 2308)。

一般来说，我们存储的资源记录都应该是明确有效的，但也有一些根本不存在的内容。如果有人问我，你带充电器了吗，我找了一下书包，发现没有后，就会告诉对方我没有带。





那么下次再有其他人问起来，我用不着翻书包，直接告诉他我没带。对于查询的名称不存在或者数据错误的 NXDOMAIN、NODATA，BIND 也会把它们缓存起来，防止有人再去问同样的问题。Negative Response TTL 就是为这准备的。

应当为 TTL 设置一个较小的数值，建议是一分钟。遇到紧急切换 DNS 域名的时候，TTL 时间过长会拖后腿，让故障时间也随之加长。因为，在 TTL 耗尽之前，比如 Windows Shell 的客户端默认是不再重新查询的。当然是否受到 TTL 的限制，取决于客户端，比如 Linux 的 dig 就不受影响。由于我们不能控制客户端的逻辑，因此，服务器这边就要控制好 TTL 的时长。当然，TTL 值也不能设置得太小，否则会增加服务器的负载。

### 9. zone 文件的生命周期

zone 文件中的 SOA 记录里面有四个生命周期。

refresh 用来设置 Slave Server 多久刷新一次去检查区域文件是否更新。这个时间不要设置得太短，如果 zone 文件的数量和尺寸比较大，过短的刷新时间可能会让 Slave Server 一直处于 reload 的过程中。

如果 refresh 失败，retry 用来设置 Slave Server 多久之后再进行重新尝试。通常，retry 应当小于 refresh。尽管这不是必需的，但反其道而行之的话，则没有任何意义。

expire 指的是过期时间，如果 Slave Server 在过期时间到来时，仍然无法联系到 Master Server 的话，则这个区域文件的数据内容将会失效。对于这个区域的查询，Slave Server 将不再应答，因为过期的信息已经不可靠了。

TTL，这里指的是 Negative Response TTL，其含义上面已经解释过了，不再赘述。

### 10. 偷懒大法

zone 文件中有很多省略的写法，有机会的话，能用就用喽。

\$ORIGIN 用来指定一个域名，当 RR 的 name 的结尾没有以点 (dot) 结束时，就会自动加上 \$ORIGIN。使用动态更新时，bind 会自动生成这个变量。默认情况下，当 name 的结尾没有点 (dot) 结束时，也会自动附加当前 zone 的域，这是由 zone 语句中的第二个字段决定的，这个字段（域名）是 zone 文件中所有数据的来源 (origin)，而且如果一个域名和来源 (origin) 相同，就会用 @ 来表示。

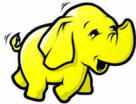
```
$ORIGIN example.com.
```

如果我要批量部署几千台服务器，服务器的主机名是 stationX，X 又和 IP 地址相关联，那么写几千个 RR 岂不是非常麻烦的事情吗？既然 IP 是已知的，那么 X 就有规律可循，何不使用 \$GENERATE 来解决这个问题呢？

\$GENERATE 的语法：

```
$GENERATE range lhs type rhs [comment]
```

\$GENERATE 的示例：



```
$ORIGIN example.com.  
$GENERATE 1-250 station$ A 192.168.0.$
```

就等价于

```
station1.example.com. A 192.168.0.1  
station2.example.com. A 192.168.0.2  
...  
station250.example.com. A 192.168.0.250
```

对于 TTL 的声明，不仅可以在第一行写入 \$TTL 为所有的 RR 定义，还可以单独为某一条 RR 定义 TTL。不过，如果有多组 RR 记录需要不同的 TTL 定义，则没有必要在每一条 RR 上面设置，只需在每一组 RR 的前面声明就可以了。例如下面这样。

```
$TTL 60 ; 1 minute  
pop      A      192.168.22.102  
smtp     A      192.168.22.101  
  
$TTL 30 ; 30 seconds  
station103 A      192.168.0.73  
email     CNAME   mail  
mail      A      192.168.22.100  
www       A      192.168.0.65  
          A      192.168.0.62  
          A      192.168.0.61
```

通过上面这个例子，我们同时还发现了另外一个省略技巧：当一个名字有多条 A 记录时，我们可以只写一个 name。

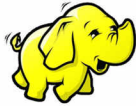
## 11. 修改配置文件后是否需要重启 named 服务

重启 named 服务可不是一个好主意。named 服务在开启监听之前，需要先检查配置文件和 zone 文件的语法格式。任何错误都将导致 named 启动失效，然后就会殃及整个线上生产系统。所以，永远都不要轻易地去重启 named 服务。

正确的做法是采用 rndc 的方式管理和控制 BIND。从 BIND 8.2 开始，ISC 引入了控制通道的方式来向 UNIX Domain Socket 发送控制消息。BIND 8 中使用 ndc，通过 TCP 953 端口控制 BIND。到了 BIND 9 之后，ndc 变成了 rndc。

rndc 需要 rndc-key 共享密钥才能执行。借助 rndc-confgen 可以生成相关的信息来指导用户如何配置。新版本的 BIND 9 默认不写任何配置也是可以在本地执行 rndc 命令的。named 启动时会自动创建一个含有密钥的 rndc.key 文件，这个密钥停止服务后会被删除，但重新生成时密钥也不会改变。与此不同的是，使用 rndc-confgen 生成的密钥是随机的。执行 rndc-confgen 后，按照提示完成配置，此时 rndc 不能直接执行，需要使用 -k 来指定 keyfile 文件。

我们可以通过 rndc 去重载配置文件或者 zone 文件。当文件出现问题的时候，最多就是



新版本的内容不生效而已，但不会影响到现有 named 服务的正常运行。当然，最好的方式还是在重载之前，对配置文件或者 zone 文件进行语法检查。

- ❑ rndc reload 用于重载所有的配置文件和 zone 文件。
- ❑ rndc reload zone 用于重载指定的 zone 文件。
- ❑ rndc refresh zone 用于 Slave Server 刷新指定的 zone 文件。
- ❑ rndc retransfer zone 用于强制区域传输（无视 SN 的变化）。

## 12. 如何检查 named 的服务状态和版本信息

使用命令 rndc status 可以显示很多有用的状态信息，包括版本、调式级别、执行区域传输的数量以及日志是否开启等。

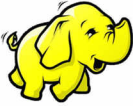
```
[root@station103 ~]# rndc status
version: 9.8.2rc1-RedHat-9.8.2-0.17.rc1.el6_4.6
CPUs found: 32
worker threads: 32
number of zones: 17
debug level: 0
xfers running: 0
xfers deferred: 0
soa queries in progress: 0
query logging is OFF
recursive clients: 0/0/1000
tcp clients: 0/100
server is up and running
```

## 13. 如何查询 DNS 统计数据

没有必要单独部署一个监控脚本，使用命令 rndc stats 就可以按照时间段获取到所有有关的统计信息。命令执行完后，不会在屏幕上输出任何内容，有关的结果会被保存到 /var/named/chroot/var/named/data/named\_stats.txt 这个文件当中。

```
+++ Statistics Dump +++ (1494558367)
++ Incoming Requests ++
      997 QUERY
      1 STATUS
++ Incoming Queries ++
      982 A
      1 TXT
      1 AAAA
++ Outgoing Queries ++
[View: default]
      9 A
      4 NS
      5 AAAA
[View: _bind]
++ Name Server Statistics ++
      998 IPv4 requests received
```

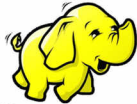




```
      2 TCP requests received
    997 responses sent
    981 queries resulted in successful answer
    978 queries resulted in authoritative answer
      5 queries resulted in non authoritative answer
      1 queries resulted in nxrrset
      1 queries resulted in NXDOMAIN
      5 queries caused recursion
      1 duplicate queries received
++ Zone Maintenance Statistics ++
++ Resolver Statistics ++
[Common]
[View: default]
    18 IPv4 queries sent
    18 IPv4 responses received
      1 NXDOMAIN received
      4 IPv4 NS address fetches
      4 IPv6 NS address fetches
      9 queries with RTT 10-100ms
      3 queries with RTT 100-500ms
      1 queries with RTT 500-800ms
      4 queries with RTT 800-1600ms
      1 queries with RTT > 1600ms
[View: _bind]
++ Cache DB RRsets ++
[View: default]
    39 A
      7 NS
      2 CNAME
    33 AAAA
      1 RRSIG
      1 !AAAA
      1 NXDOMAIN
[View: _bind (Cache: _bind) ]
++ Socket I/O Statistics ++
    20 UDP/IPv4 sockets opened
      3 TCP/IPv4 sockets opened
      1 TCP/IPv6 sockets opened
    18 UDP/IPv4 sockets closed
    30 TCP/IPv4 sockets closed
    18 UDP/IPv4 connections established
    31 TCP/IPv4 connections accepted
++ Per Zone Query Statistics ++
--- Statistics Dump --- (1494558367)
```

美中不足的是，这个 dump 文件的时间戳可读性不太好，我们可以通过下面这段脚本做一下转换。

```
#!/bin/bash
DATADIR=/var/named/chroot/var/named/data
```



```
DATAFILE=named_stats.txt
cp -f $DATADIR/$DATAFILE $DATADIR/$DATAFILE.new
for i in `awk '/Statistics Dump/ {split($0,a,"[()]");print a[2]}' $DATADIR/
$DATAFILE`
do
    sed -i "s/$i/$ (date -d @$i) /g" $DATADIR/$DATAFILE.new
done
```

这段脚本的目的就是调整时间格式，把秒数变成可读的时间。

\$DATADIR/\$DATAFILE 的内容

```
+++ Statistics Dump +++ (1494558367)
...
--- Statistics Dump --- (1494558367)
```

\$DATADIR/\$DATAFILE.new 的内容

```
+++ Statistics Dump +++ (Fri May 12 11:06:07 CST 2017)
...
--- Statistics Dump --- (Fri May 12 11:06:07 CST 2017)
```

## 14. 如何调试 DNS

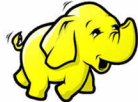
有时日志给出的信息量太少，内容输出往往只是结果或者原因，但具体细节都被隐藏了。为了进一步分析问题，我们需要使用命令 `rndc trace` 来获取更详细的调试信息。如果不指定调试级别，默认是 `Level1`。BIND 9 的调试级别清单如表 8-1 所示。

表 8-1 BIND 9 的调试级别清单

| Debug Level | 描 述                             |
|-------------|---------------------------------|
| 1           | 涉及基本的管理维护操作、Notify 消息、查询 NS 记录等 |
| 2           | 多播消息                            |
| 3           | 日志活动、DNSSEC 验证、TSIG 检查          |
| 4           | AXFR 事件                         |
| 5           | 视图的使用情况                         |
| 6           | 区域外的传送及查询信息                     |
| 7           | 日志增加、删除及区域传送返回字节数               |
| 8           | 动态更新、区域传送资源记录                   |
| 10          | 区域计时器的活动                        |
| 20          | 更新出发的区域计时器刷新                    |
| 90          | 低级别任务                           |

## 15. 使用 chroot 确保安全

任何软件都不可避免地存在安全漏洞。恶意攻击者会利用溢出或者逻辑错误先获取某一个服务的权限，然后利用现有资源去尝试进一步提权。这样的攻击方式很容易危害整个



系统的安全，而 chroot 可以最大化地减少这种可能性。使用 bind-chroot 几乎是所有 BIND 管理员的安全共识。但是 bind-chroot 默认把目录定向到 /var/named/chroot/，而这个目录空间的挂载点通常就是根目录。如果我们希望把 BIND 的数据挂载到一个单独的空间里，和根目录完全隔离开来，只要修改 /etc/sysconfig/named 文件中的 ROOTDIR=/var/named/chroot，再将它设置成你所希望的路径就可以了。

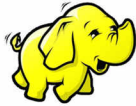
## 16. 如何规划目录

在目录规划上，我用自己的测试环境作为范例。我们基本遵循原有的目录设计结构，仅做了一些小范围的调整。正如第 7 章中所讲的那样，我们尽可能采用层级嵌套的方式收敛配置文件，而不是把一大堆的东西都平铺到一个目录当中去。这是符合现代系统管理的主流风格。

在 named.conf 文件中，我们只配置 option、channels 和 logging，ACL 和 View 通过 include 语句完成对外部配置文件的引用。ACL 和 View 同样只引用实例文件的名称，在实例文件中再完成详细的配置。同理，zone 文件、日志文件和统计数据文件也是这样处置。这种规划方式大有裨益，它实现了在阅读配置文件时尽可能地排除干扰，在修改配置文件时最小化影响范围，不会因为局部错误而伤及无辜。这个思路，我想大家在第 7 章当中已经有所体会了。

```
/var/named/chroot/           // 用于存放配置文档及执行脚本的总目录
|-- etc/                     // 用于存放所有配置文件的总目录
|   |-- named.conf           // 主配置文件
|   |-- named/               // 用于存放其他配置文件的目录
|       |-- acl/             // 用于存放 ACL 配置的目录
|           |-- acl.conf     // ACL 引用配置文件
|           |-- xxx.acl      // 一个 ACL 实例文件
|           |-- ...          // 还有很多的 ACL 实例文件
|       |-- view/            // 用于存放 View 配置的目录
|           |-- view.conf    // View 引用配置文件
|           |-- xxx.view     // 一个 View 实例文件
|           |-- ...          // 还有很多的 View 实例文件
|   |-- pki/                 // 用于存放 View 配置的目录
|       |-- dnssec-keys/     // 用于存放密钥文件的目录
|       |-- ...              // 其他原生的配置文件或者密钥文件
|-- var/
|   |-- named/               // 用于存放资源数据的总目录
|       |-- zone/            // 用于存放正向解析数据的目录
|       |-- rev/             // 用于存放反向解析数据的目录
|       |-- data/            // 用于存放统计数据的目录
|   |-- log/                 // 用于存放日志的总目录
|       |-- query_log/       // 用于存放查询日志的目录
|       |-- server_log/      // 用于存放服务日志的目录
|       |-- security_log/    // 用于存放安全日志的目录
|       |-- ...              // 还可以划分其他类型的日志
|-- ...                      // 其他相关目录
```





## 17. CNAME 不是别名

“嘿，给我配置一个 CNAME 别名。”

在日常工作中，我们常常会听到这样的说法。刚开始学习 BIND 的很长一段时间里，我都误以为 CNAME 记录就是别名记录，但又想不明白，C 好像和“别名”这个词根本就是八竿子都打不着。别名的英文明明是 alias 或 byname 嘛。

其实 CNAME 全称是 Canonical Name，CNAME 记录的域名是规范名称，而要点指向 CNAME 的域名才是别名。所以，你不是要配置一个 CNAME 别名，而是要将一个别名指向一个 CNAME。

## 18. 轮询调度

如果在 BIND 中同一个域名设置了多条 A 记录，则 BIND 会将对这些地址进行轮询 (Round Robin)。在等效网络中，例如用于 yum repo 的 HTTP 服务，Round Robin 确实带来了一些有益的东西。但是它并不是负载均衡，而是负载分配。它只是根据请求来回应而已，对后端服务器的表现情况根本一无所知。所以，请不要在生产中依赖 Round Robin 去取代你的网络负载均衡设备。另外，即使是在等效网络里，为了避免分配不均的情况，也应通过 rrset-order 来设置轮询的策略。

rrset-order 的语法如下。

```
options {  
    rrset-order {  
        order_spec;  
        [ order_spec; ... ]  
    }  
};
```

order\_spec 的定义如下。

```
[ class string ] [ type string ] [ name quoted_string ] order ordering_string
```

Class 和 Type 的默认值是 Any，Name 默认值为 \*，ordering\_string 有三个策略。

- ❑ fixed：以资源文件中的记录顺序返回，这种方式使得大多数客户端选择第一条而忽略其余的记录。
- ❑ random：以随机的方式返回，随机也难以保障平均，尤其是在 A 记录条目很少的情况下，这种不公平的现象更容易出现。
- ❑ cyclic：以环的顺序返回，例如 1-2-3、2-3-1、3-1-2 这种方式，就像用手捻佛珠那样，每次返回一个，往复交替。这是我们建议采取的策略。

1) 统一策略示例：

```
options {  
    rrset-order { order cyclic; };  
};
```

## 2) 多重策略示例:

```
options {
    rrset-order {
        class IN type A name "random.example.com" order random;
        order cyclic;
    };
};
```

另外, Round Robin 不支持 CNAME, 那样做是错误的。BIND 9.1 之后都会审查这个错误的问题。不幸的是, rndc 却并不会察觉这个错误, 只是客户端查询时无法返回结果而已。而 named 服务在启动的时候会返回 multiple RRs of singleton type 的错误。

```
www1    A      192.168.100.1
www2    A      192.168.100.2
www3    A      192.168.100.3
www     CNAME  www1.example.com.
www     CNAME  www2.example.com.
www     CNAME  www3.example.com.
```

## 19. 转发器

任何一台 DNS Server 都无法解决所有的域名解析, 所以当遇到 DNS Server 自己没办法处理的问题时, 向别人询问是最好的解决办法。转发器 (Forwarder) 负责解析不属于自己负责的区域的地址。内网 DNS Server 设置转发器时要防止形成这种  $A \rightarrow B \rightarrow C \rightarrow A$  的转发环路。企业 (非 ISP) 的公网 DNS Server 只负责自己本域内的名称解析, 一般不应再配置转发器。而内网 DNS Server 的转发配置最多两跳, 第一跳是从内网到边界 DNS Server, 第二跳是从边界 DNS Server 再到公网。尽可能不要使用自己的公网 DNS Server 来充当边界 DNS Server。确保内网和外网的 DNS 解析是分离的。

## 20. 日志、通道和类别

日志, 曾经是我最不愿意研究的东西, 总是觉得麻烦。一来语法复杂、晦涩难懂, 二来概念混乱, 剪不断理还乱。早期我曾经把日志这个地方删掉, 反正也不影响运行。后来我认真研读了 BIND 手册, 才知道 logging 的重要性。

```
logging {
    [ channel channel_name {
        ( file path name
          [ versions ( number | unlimited ) ]
          [ size size spec ]
          | syslog syslog_facility
          | stderr
          | null );
        [ severity ( critical | error | warning | notice | info | debug [level
          ] | dynamic ); ]
        [ print-category yes or no; ]
        [ print-severity yes or no; ]
```

```

        [ print-time yes or no; ]
    }; ]

    [ category category_name {
        channel_name ; [ channel_name ; ... ]
    }; ]
    ...
};

```

通道 (Channel) 是为日志创建一个管道, 意味着日志将以什么形式输送出来。通道的命名使用关键词 `channel` 定义。这个名字可以随便起, 但是我建议大家最好和通道及类别关联起来。这样做一目了然, 根据 `channel name` 就可以确定这个 `channel` 是干什么用的, 将产生什么数据, 并以什么形式留存。serverity 就是 BIND 的日志级别, 相当于 rsyslog 中的 priority。

通道共有四类。

1) file channel 用于在本地产生日志文件。

2) syslog channel 将日志发送到 rsyslog, 由 rsyslog 的配置决定日志是在本地保存, 还是转发到远程的日志采集服务器上。使用 syslog channel 就要用到 syslog 语句。syslog 语句可指定 named 把日志发给 rsyslogd 时以什么身份发送出来, 默认是 daemon。这就是我们什么也不配置, 就可以在 /var/log/messages 中看到 named 日志的原因。除了 daemon, 我们也可以选择 local0-local7。

3) stderr channel 将日志以标准错误输出方式输送出来。

4) null channel 将你不需要的无用信息存储在这里。

类别 (Category) 就是消息分类。这个不要和 syslog 语句里面的 syslog\_facility 搞混了。syslog\_facility 是指日志到了 rsyslogd 那边是什么类别, 而 Category 是 BIND 消息的类别。

默认 syslog channel 设置 syslog\_facility 为 daemon, 表示从 syslog 发出去的日志将以 daemon 的身份出现, 根据 rsyslog.conf 文件配置, 我们知道 daemon 的日志都会记录到 /var/log/messages 日志当中, syslog\_facility 设定好后只会遵从 rsyslog.conf 的配置决定写入哪个系统日志文件当中, 但不管怎么写, 所有的 BIND 消息都是记录在同一个文件中的。

Category 则是用来拆分 BIND 消息。Category 支持很多种类别。例如, client 是用于处理客户端请求的, config 则是和配置文件的分析相关的内容。如果没有消息分类, 这些 BIND 消息将一股脑儿地全部写入到一起。这样对于分析日志是没有帮助的。

默认情况下, 如果没有为某个类别指定通道, 那么它们都将属于 default 类别。这时的 default 类别就代表所有类别。关于这一点, BIND 8 和 BIND 9 有所区别。对于有些根本不属于任何分类的消息, BIND 8 的 default 类别也会收容这些“不合群”的消息, 但 BIND 9 的 default 类别就不会这样做。

下面是一个 logging 的示例。



```

logging {
    channel syslog_client {
        syslog local0
        serverity info;
        print-category yes;
        print-serverity yes;
        print-time yes;
    }
    channel file_config {
        file "config.log" version 3 size 512K
        serverity warning;
        print-serverity yes;
    }
    category client { syslog_client };
    category config { file_config };
}

```

## 21. DNS Notify

尽管 Slave Server 会根据 refresh 的设定定期更新数据文件，但是这个机制是不够的，因为很多紧急的情况需要 Master Server 主动、及时地通知 Slave Server 变更数据文件。DNS Notify 用于通告该区域所有的 Slave Server。重启 named 服务，或者增加 SN 并重载一个 zone 文件（包括动态更新）都会触发 DNS Notify。DNS Notify 是 Master Server 主动发起的同步保障手段。

## 22. 动态更新

RFC 2136 提出动态更新（Dynamic Domain Name Server, DDNS）的功能。该功能允许经过授权的 Updater 在区域内请求增加或删除资源记录。Updater 通过检索 zone 文件的 NS 记录找到 DNS 服务器，如果这个 DNS 服务器不是 Master，那么它将无法修改 zone 文件的内容，它需要将这个更新请求继续向上转发，这个过程称之为更新转发。

动态更新的好处在于：无须对 zone 文件进行大量的反复修改，而且不必担心因为人为或者程序修改而造成的误操作。nsupdate 可以有效地负责检查。nsupdate 提交要比手工修改靠谱得多。另外，动态更新需要严格限制并且使用安全的 key。

常用的 nsupdate 方法如下。

```
nsupdate [-l] -d -k KEYFILE UPDATE_FILE
```

- ❑ -d 代表 debug 模式，用于跟踪更新过程。
  - ❑ -k 是基于共享加密生产 TSIG key，对动态更新的合法性进行签名验证。以确认 Updater 的真实身份，防止其他人通过攻击 Updater 下线并篡改 IP 地址等手段来冒充。
  - ❑ -l 代表在本地进行 nsupdate 操作，而且本地的 keyfile 将覆盖 -k 指定的文件。
- 更新文件的内容格式如下。

```
server {servername} [port]
```

```

local {address} [port]
zone {zonename}
class {classname}
ttl {seconds}
update add {domain-name} {ttl} [class] {type} {data...}
update delete {domain-name} [ttl] [class] [type] [data...]

```

为了更加严谨，在执行 update 之前，可以增加如下约束条件。

```

prereq nxdomain {domain-name}
// 请求执行之前，约束指定的域名内不存在任何资源记录，通常是尚未创建这个 zone
prereq yxdomain {domain-name}
// 请求执行之前，约束指定的域名至少存在一条资源记录
prereq nxrrset {domain-name} [class] {type}
// 请求执行之前，约束指定的域名 + 类 + 类型不存在，class 不指定，默认为 IN
prereq yxrrset {domain-name} [class] {type}
// 请求执行之前，约束指定的域名 + 类 + 类型不存在，class 不指定，默认为 IN

```

简单的动态更新可以不需要前面的那些全局变量。我们直接提交 update add 或者 update delete 也是可以的。注意一行提交一个记录，最后一行以 send 语句或者一个空行结束，用于提交更新请求。更新结束后，会在区域文件处生成一个和 zone 文件同名的并且以 .jnl 结尾的日志数据文件。

例如，我们可以这样提交。

```

update add pop.example.com. 60 IN A 182.17.22.102
update add smtp.example.com. 60 IN A 182.17.22.101
send

```

动态更新前，我们的 zone 文件内容如下。

```

$TTL 30
example.com      IN SOA  station103.example.com. root.example.com. (
                                14      ; serial
                                10800   ; refresh (3 hours)
                                3600    ; retry (1 hour)
                                604800  ; expire (1 week)
                                60      ; minimum (1 minute)
                                )
                                IN NS   station103.example.com.

station103 IN A      192.168.0.73
email      IN CNAME  mail
mail       IN A      192.168.22.100
www        IN A      192.168.0.65
           IN A      192.168.0.62
           IN A      192.168.0.61

```

动态更新后，我们发现 zone 文件发生了一些变化。

```

$ORIGIN .
$TTL 30 ; 30 seconds
example.com      IN SOA  station103.example.com. root.example.com. (
    14          ; serial
    10800       ; refresh (3 hours)
    3600        ; retry (1 hour)
    604800      ; expire (1 week)
    60          ; minimum (1 minute)
    )

                IN NS   station103.example.com.

$ORIGIN example.com.

$TTL 60 ; 1 minute
pop      IN  A      192.168.22.102
smtp     IN  A      192.168.22.101

$TTL 30 ; 30 seconds
station103 IN A      192.168.0.73
email     IN  CNAME  mail
mail      IN  A      192.168.22.100
www       IN  A      192.168.0.65
          IN  A      192.168.0.62
          IN  A      192.168.0.61

```

动态更新后，直接打开 zone 文件是没有办法看到更新后的内容的。我们需要重启 named 服务，但是这个方式太蠢了。另外一个办法就是 rndc reload。动态更新直接使用 rndc reload 也是看不到内容的，要使用 rndc reload example.com 才可以。但是，执行这个命令时你会发现，rndc 告知我们这是一个动态区域，不能 reload。这里，需要先执行 rndc freeze example.com 暂停动态更新，然后再执行 rndc reload example.com，最后再次执行 rndc thaw example.com 重新恢复动态更新。此时就可以看到更新后的文件内容了。

BIND 8 中，DNS 服务器每 5 分钟或者累计 100 次更新（以先到达为准）才增加 zone 的 SN。在 BIND 9 中，动态更新的序号每更新一条就修改一次，所以如果是大批量的动态更新，SN 就不适合使用 YYYYMMDDNN 的方案。

### 23. 超时问题

如果客户端请求一个域名解析的时候发现无法连接 DNS 服务器是一件非常恼人的事情。这比查询不到地址还令人讨厌。如果不知道，直截了当地告诉你不知道也就算了。可恶的是，你问了一个问题，结果他不搭理你，等了很久也不回答，直到你失去耐心离开为止。

从 BIND 8.2 开始，引入了几个新的选项，需要在 /etc/resolv.conf 进行配置。

timeout：超时时间，单位是秒，默认值为 5，最大可以设置为 30。



attempt：尝试次数，BIND 8.2 之前的默认值为 4，BIND 8.2 之后的默认值为 2，最大值为 5。

遇到无法连接 DNS 服务器时，总的超时时间和上述两个 option 的取值有关。超时时间配置如表 8-2 所示。

表 8-2 超时时间配置

| 重试次数          | 第一台 DNS 服务器 | 第二台 DNS 服务器   | 第三台 DNS 服务器   |
|---------------|-------------|---------------|---------------|
| Retry 0       | 5           | $2 \times 5$  | $3 \times 5$  |
| Retry 1       | 10          | $2 \times 5$  | $3 \times 3$  |
| Retry 2       | 20          | $2 \times 10$ | $3 \times 6$  |
| Retry 3       | 40          | $2 \times 20$ | $3 \times 13$ |
| Total Timeout | 75          | 80            | 81            |

如果客户端指向的 DNS Server 有多台且至少有一次 retry，在所有的 DNS Server 都联络不上的情况下，客户端的请求顺序是这个样子的：先访问第一台 DNS Server，超时过后接下来就是第二台，然后是第三台，当所有的 DNS Server 都请求完一轮后，再开始新一轮的重试。有点儿像发扑克牌不是吗？

这样做很合理，因为如果第一台不理你，没有必要急着再去问一遍，应该先拜访其他服务器，实在不行到下一轮再去搭理它。

那么，这个 Total Timeout 是怎么计算的呢？

我们首先看横行代表 DNS 的数量，纵列代表 attempt，我们以 BIND 8.2 版本之前的默认值为例，timeout=5，attempt=4（有三次 retry），我们看在只有一台 DNS 服务器的时候。每一次 retry 的超时时间是 timeout 的  $2^{\text{retry}}$  倍，第三次 retry 就是  $2^3=8$ 。多台服务器就不是这样去计算了，不然 Total Timeout 岂不是非常得长？

多台的算法是  $\text{Floor}(\text{timeout} \times 2^{\text{retry}} / N)$ ，Floor 代表向下取整， $N$  代表服务器的数量。例如当出发第二次 retry 时，超时时间就是  $\text{Floor}(5 \times 2^2 / 3) = 6$ 。

还有一个很重要的 option 叫作 rotate，如果没有它，客户端在传统解析时，永远是按照 nameserver 的顺序去查询，除非第一台服务器挂了，否则备选的 DNS Server 完全处于 Standby 状态，而首选 DNS Server 则会累个半死。

rotate 的引入解决了这种尴尬，它可以轮流向 nameserver 发起请求。但是这个效果并不是所有的客户端都领情，例如程序 ping 就是这样。因为 ping 每次都会初始化解析器，所以 rotate 都会被重新定向回第一个 nameserver。但不管怎样，有总比没有强。为了测试客户端应用是否支持，可以构建两个独立的 DNS Server 隔离测试环境，针对同一个域名分别采用不同的域名配置来测试。当然，最好的办法还是使用本章后面即将讲到的 Anycast DNS 方案。

鉴于生产环境内网的稳定性和性能，没有必要把 timeout 和 attempt 设置得太高，Any-cast 技术或者其他冗余手段可以保证 DNS 服务的可靠性，再用 retry 就显得有些多此一举了。在生产环境中建议大家如下设置。

```
option rotate
options timeout:2
options attempts:1
```

## 24. 权威应答与非权威应答

曾经有个人问我：为什么我的应答都是非权威的？难道我用了假的 DNS 服务了吗？

假设你查询的域名并不是你的 DNS 服务器负责解析的域名，那么它将向该区域权威服务器发送查询请求，此时本地 DNS 服务器会将结果缓存下来再返回给用户，所以用户得到的已经是一个来自于本地 DNS 的缓存，自然就是非权威的。

我以本章中使用的 example.com 区域为例。第一次在 whois 站点上查询可以得知它的 Owner 是 PRIVACYDOTLINK CUSTOMER 401471，这不是一个真实的名字，从字义上我们知道它掩盖了用户的身份，但它是确实存在的，而非像我们想象的那样只是一个 example。相反，我的测试环境里也使用了 example.com 区域，而在向这台 192.168.0.73 的这台服务器查询 example.com 的时候，给出的却是权威应答。不过这个权威应答是我伪造的，反而非权威倒是真实的。

下面介绍如何判断是否是权威应答。假如是权威应答，dig 返回中会有 AUTHORITY SECTION 字段，nslookup 则没有任何显示。如果是非权威应答，dig 返回中就没有 AUTHORITY SECTION 字段，而 nslookup 则会显示 Non-authoritative answer。

```
[root@station101 etc]# dig @192.168.0.73 www.example.com

;<<>> DiG 9.8.2rc1-RedHat-9.8.2-0.17.rc1.el6_4.6 <<>> @192.168.0.73 www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->HEADER<<- opcode: QUERY, status: NOERROR, id: 13707
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 1, ADDITIONAL: 1

;; QUESTION SECTION:
;www.example.com.                IN      A

;; ANSWER SECTION:
www.example.com.                9       IN      A      1.2.3.62
www.example.com.                9       IN      A      1.2.3.65
www.example.com.                9       IN      A      1.2.3.61

;; AUTHORITY SECTION:
example.com.                    6       IN      NS      station103.example.com.

;; ADDITIONAL SECTION:
```

```
station103.example.com. 6      IN      A      192.168.0.73
```

```
;; Query time: 0 msec
;; SERVER: 192.168.0.73#53 (192.168.0.73)
;; WHEN: Sun May 14 20:38:18 2017
;; MSG SIZE rcvd: 122
```

```
[root@station101 etc]# nslookup
```

```
> www.example.com
```

```
Server:      192.168.0.73
```

```
Address:     192.168.0.73#53
```

```
Name:   www.example.com
```

```
Address: 1.2.3.62
```

```
Name:   www.example.com
```

```
Address: 1.2.3.65
```

```
Name:   www.example.com
```

```
Address: 1.2.3.61
```

```
>
```

```
[root@station100 ~]# dig www.example.com
```

```
;<<>> DiG 9.8.2rc1-RedHat-9.8.2-0.17.rc1.el6_4.6 <<>> www.example.com
```

```
;; global options: +cmd
```

```
;; Got answer:
```

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26178
```

```
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
```

```
;; QUESTION SECTION:
```

```
;www.example.com.      IN      A
```

```
;; ANSWER SECTION:
```

```
www.example.com.      29650   IN      A      93.184.216.34
```

```
;; Query time: 2 msec
```

```
;; SERVER: 202.106.0.20#53 (202.106.0.20)
```

```
;; WHEN: Sun May 14 20:37:50 2017
```

```
;; MSG SIZE rcvd: 49
```

```
[root@station100 ~]# nslookup
```

```
> www.example.com
```

```
Server:      202.106.0.20
```

```
Address:     202.106.0.20#53
```

```
Non-authoritative answer:
```

```
Name:   www.example.com
```

```
Address: 93.184.216.34
```

```
>
```

所以，权威与否只是证明查询的域名是否是查询服务器所管理的而已。对于外部域名



解析我们得到的都是非权威的应答。

#### 注意：

BIND 9 以前的早期版本，第一次查询结果是权威的，以后用户再次查询才是缓存的答案。

### 25. 申请域名也有学问

合法的域名包括字母（不区分大小写）、数字和中横线（-）。申请域名时应当注意三点：不要使用非法字符，不要乱用缩写，不要滥用中横线。其中乱用缩写是域名使用问题当中最为严重的。请看下面这个例子。

```
dns-wm.example.com
dns-ws.example.com
dns-lm.example.com
dns-ls.example.com
```

我很想做个调查问卷问问大家，谁能告诉我 wm、ws、lm、ls 这些字符代表什么意思？反正我是没看出来，所谓的 w 和 l 是外、里的拼音首字母，m 和 s 是主、从的英文首字母。当我第一眼看到它们的时候，想到的是另外四个词。你可以找个拼音输入法来试一试就知道了。这四个不明就里的域名缩写是给某个产品用的，我觉得出现这种问题实在是非常不应该。我宁愿使用下面这四个名字。

```
exmaster-dns.example.com
exslave-dns.example.com
inmaster-dns.example.com
inslave-dns.example.com
```

域名这种东西最怕用拼音，尤其是拼音缩写。如果你要用缩写，那也请用英文缩写，而且这个缩写一定是众所周知、约定俗成的，至少也应该是大家公认的内部写法。

域名长一点儿没有关系，但是一定要把意思表达清楚。

什么时候用中横线呢？起域名时，如果在充分表达含义后发现字符串变得非常冗长，而且又无法使用恰当的缩写，那么中横线可以起到分隔效果，增加域名的可读性。

### 26. dig 的用法

使用 +trace 参数查询时，本地 DNS Server 将从 Root Server 开始，执行一次完整的迭代查询，它会将每一个路径上遇到的 DNS Server 都列出来。但是，你要不指望用它去跟踪一个递归查询。

默认情况下，dig 使用的是客户端配置的本地 DNS Server。使用 @ 以后就可以指定任意一台 DNS Server 请求域名解析，根据返回的查询时间，你可以把客户端的 DNS Server 更换成响应速度更快的那一台。

dig 的返回信息太多怎么办？我们使用下面这套组合拳就好了。

```
[root@station103 ~]# dig +nocmd +nocomment +noquestion +nostat www.example.com
www.example.com.          69396      IN         A          93.184.216.34
```

或者还可以更加简约一些。

```
[root@station103 ~]# dig @8.8.8.8 +short www.example.com
93.184.216.34
```

如果一次要查询很多个域名，可以使用 -f file。注意：文件中每一行都代表一个域名，所以千万不能出现空行。

## 8.3 Anycast DNS 在多数据中心中的应用

DNS 作为基础服务，一旦出现故障将导致非常严重的后果。仅仅采用简单的主从架构不足以保障 DNS 可靠性。因为传统的一主多从结构只能确保服务端的可用性，但忽略了客户端配置的因素。因为多个 DNS Server 的地址是不一样的，而客户端的指向中包含多个 DNS Server 时，不管是基于什么策略，客户端都会先查询某一台 DNS Server。假设首选 DNS Server 出现了故障，那么用户至少要等到查询超时之后，才会转向另外一台 DNS Server。

### 8.3.1 什么是 Anycast

Anycast 最初是在 RFC1546 中提出并定义的。它允许把一个单播地址分配给多个接口，路由器会根据度量值自动选择最佳路径，将发送给该地址的报文投递到最近的接口上，如图 8-5 所示。这样一来，Anycast 使得应用层服务具有更强的透明性，多个服务节点或者网络接口可以共享一个 IP 地址。路由选择一方面缩短了响应时间，另一方面也实现了冗余和均衡的效果。此外，它还有有效地降低了网络拥塞，和 DDoS 攻击对网络造成的影响，因为当 Anycast 组内的某些成员出现响应延迟或者根本无法响应时，报文会被转发到响应更快的其他成员上面。

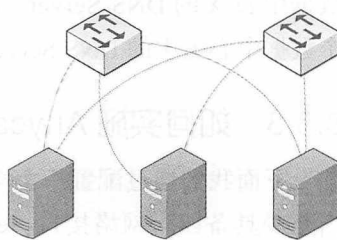


图 8-5 Anycast

使用 Anycast 目前唯一的限制是不允许共享的单播地址发起请求，因为请求响应不一定能返回给发起的那个接口。不过，Anycast 技术非常适合无状态的应用服务场景（例如 HTTP/DNS），访问它们对于客户端来说完全是透明的。

### 8.3.2 如何构建 DNS over Anycast

同城多数据中心多是采用不同电信供应商的服务，用户流量分割依据服务商的不同会被划分到不同的数据中心去。图 8-6 给读者展示了如何基于多数据中心在内部网络中构建一个 Anycast DNS。我们有 A、B 两个数据中心，通过视图设置，将电信用户指向 A，联通

用户指向 B。当数据中心 A 内部的应用服务器节点出现故障时，通过修改视图，将电信用户组加入到数据中心 B 的视图中来，实现流量切换。

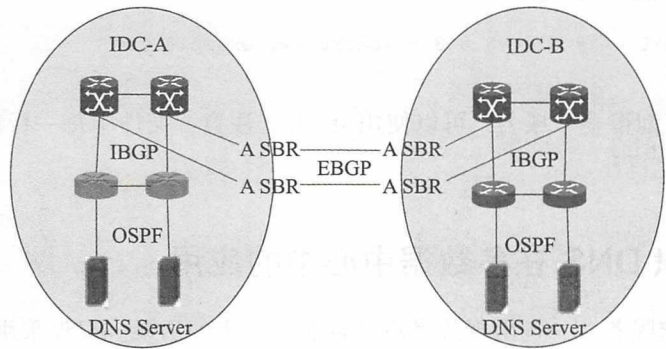


图 8-6 Anycast 技术在多数据中心的应用

如果数据中心 A 的内网 DNS Server 故障，利用 Anycast 技术可以让 A 中其他的 DNS Server 来响应查询请求。如果 A 中所有的 DNS Server 都出现了故障，那么 Anycast 允许 A 中所有的应用服务器使用 B 中的 DNS Server 完成解析。

两个数据中心的 ASBR 通过 EBGP 分别将本地 DNS Server 通告给对端，并将对端的 DNS Server 发布到本地的 OSPF 进程，从而使得两个数据中心的 DNS Server 互为冗余。当数据中心 A 的 DNS Server 出现故障时，该数据中心内网的 DNS 流量查询将被引入另外一个数据中心当中的 DNS Server 上去。

### 8.3.3 如何实施 Anycast DNS

下面我们通过配置一个实例节点，来说明如何实施 Anycast DNS。作为 DNS Server 应当至少具备四个网络接口。em3 和 em4 相互之间做了 Bonding，em1 和 em2 分别接入本地两台运行了 OSPF 的三层交换机当中。em1 和 em2 属于 Anycast DNS 网络中的成员，而 Bonding 的网卡仅用于日常登录管理，两个网络应当相互区分开来。

为了便于说明，首先我们需要定义变量，如表 8-3 所示。

表 8-3 定义变量

| 变量 \ 网络接口   | em1      | em2      |
|-------------|----------|----------|
| INTERFACE   | IF1      | IF2      |
| IP ADDRESS  | IP1      | IP2      |
| NETWORK     | NET1     | NET2     |
| GATEWAY     | GATEWAY1 | GATEWAY2 |
| ROUTE TABLE | TABLE1   | TABLE2   |



接下来，定义如下路由表和路由规则。

### (1) 路由表声明

```
/etc/iproute2/rt_tables:
# reserved values
255    local
254    main
253    default
0      unspec
# local
#1     inr.ruhep
101    $TABLE2
100    $TABLE1
```

### (2) 路由规则定义

```
ip route add $NET1 dev $IF1 src $IP1 table $TABLE1
ip route add default via $GW1 table $TABLE1
ip route add 127.0.0.0/8 dev lo table $TABLE1
ip rule add from $IP1 table $TABLE1
```

```
ip route add $NET2 dev $IF2 src $IP2 table $TABLE2
ip route add default via $GW2 table $TABLE2
ip route add 127.0.0.0/8 dev lo table $TABLE2
ip rule add from $IP2 table $TABLE2
```

### (3) 路由规则在配置文件中的具体实现

为了保证路由规则定义重启后继续生效，我们建议将其放置在如下配置文件中。

```
/etc/sysconfig/network-scripts/route-em1:
$NET1 dev $IF1 src $IP1 table $TABLE1
default via $GW1 table $TABLE1
127.0.0.0/8 dev lo table $TABLE1
```

```
/etc/sysconfig/network-scripts/route-em2:
$NET2 dev $IF2 src $IP2 table $TABLE2
default via $GW2 table $TABLE2
127.0.0.0/8 dev lo table $TABLE2
```

```
/etc/sysconfig/network-scripts/rule-em1:
from $IP1 table $TABLE1
```

```
/etc/sysconfig/network-scripts/rule-em2:
from $IP2 table $TABLE2
```

现在，我们来实际配置一个示例。按照图 8-6 所示，我们假设了一个配置环境。环境说明如表 8-4 所示。

假设 Anycast 地址是 192.168.0.1、192.168.0.2。还有另外两个网卡做了 Bonding，但由于它们仅用于日常维护工作，与 Anycast 无关，所以我们在这里忽略关于 Bonding 的所有

配置。我们以节点 stationA-100.example.com 为例，具体的 TCP/IP 设置如表 8-5 所示。

表 8-4 环境说明

| IDC | Hostname                 | em1           | em2           |
|-----|--------------------------|---------------|---------------|
| A   | stationA-100.example.com | 192.168.1.100 | 192.168.2.100 |
|     | stationA-101.example.com | 192.168.1.101 | 192.168.2.101 |
| B   | stationB-100.example.com | 192.168.3.100 | 192.168.4.100 |
|     | stationB-101.example.com | 192.168.3.101 | 192.168.4.101 |

表 8-5 实例 stationA-100 的网络配置

| Interface | IP Address    | Netmask       | Gateway       | Table |
|-----------|---------------|---------------|---------------|-------|
| em1       | 192.168.1.100 | 255.255.255.0 | 192.168.1.254 | ospf1 |
| em2       | 192.168.2.100 | 255.255.255.0 | 192.168.2.254 | ospf2 |

### 1. 网络配置以及路由声明

首先，配置两块网卡，并套用路由表声明和路由规则定义的模板，完成相关的配置。

```
# cat /etc/sysconfig/network-scripts/ifcfg-em1
DEVICE=em1
ONBOOT=yes
TYPE=Ethernet
BOOTPROTO=none
IPADDR=192.168.1.100
NETMASK=255.255.255.0

# cat /etc/sysconfig/network-scripts/ifcfg-em2
DEVICE=em2
ONBOOT=yes
TYPE=Ethernet
BOOTPROTO=none
IPADDR=192.168.2.100
NETMASK=255.255.255.0

# cat /etc/sysconfig/network-scripts/route-em1
192.168.1.0/24 dev em1 src 192.168.1.100 table ospf1
default via 192.168.1.254 table ospf1
127.0.0.0/8 dev lo table ospf1

# cat /etc/sysconfig/network-scripts/route-em2
192.168.2.0/24 dev em2 src 192.168.2.100 table ospf2
default via 192.168.2.254 table ospf2
127.0.0.0/8 dev lo table ospf2

# cat /etc/sysconfig/network-scripts/rule-em1
from 192.168.1.100 table ospf1
```



```
# cat /etc/sysconfig/network-scripts/rule-em2
from 192.168.2.100 table ospf2
```

## 2. 创建 Anycast 接口

在这里，我们要创建两个 Anycast 地址。Anycast 接口就是未来发布给客户端的虚拟地址，客户端会向其发起 DNS 请求。这两个 Anycast 地址没有任何区别，只是在地址这个环节上也做了冗余而已。这样一来，我们的架构从地址、本地节点到数据中心，每一个环节都具备了冗余条件。

```
# cat /etc/sysconfig/network-scripts/ifcfg-lo:0
DEVICE=lo:0
IPADDR=192.168.0.1
NETMASK=255.255.255.255
ONBOOT=yes
NAME=dns1
```

```
# cat /etc/sysconfig/network-scripts/ifcfg-lo:1
DEVICE=lo:1
IPADDR=192.168.0.2
NETMASK=255.255.255.255
ONBOOT=yes
NAME=dns2
```

## 3. 安装 quagga

quagga 是一款支持多种路由协议的软路由，可以同时支持 RIP v1、RIP v2、RIPng、OSPF v2、OSPF v3、BGP 4 等。它基于模块化设计，对每一个路由协议都使用单独的守护进程。quagga 的运行速度快、可靠性高。我们利用它实现和交换机之间的 OSPF 协商。quagga 软件包包括两个组件：zebra 和 ospf。

zebra 组件负责维护路由表。它的配置文件的详细内容如下所示。

```
# cat /etc/quagga/zebra.conf
! ----- zebra configurations -----
hostname stationA-100.example.com // 这里的主机名根据实际环境自行替换
password 8 aabbccddeeffgg // 这里的密钥根据实际环境自行替换
enable password 8 xxyyzzwwwuuuu // 同上
service password-encryption

log file /var/log/quagga/zebra.log

line vty
```

ospf 组件负责在本地启用 OSPF 协议，和交换机进行联络。它的配置文件的详细内容如下所示。

```
# cat /etc/quagga/ospfd.conf
! ----- ospf configurations -----
hostname station100.example.com
```





```

password 8 aabbccddeeffgg
enable password 8 xxyzzzwwwuuuu
service password-encryption

interface em1
ip ospf hello-interval 3
ip ospf dead-interval 12
ip ospf authentication message-digest
ip ospf message-digest-key 1 md5 hellokey
ip ospf cost 10

interface em2
ip ospf hello-interval 3
ip ospf dead-interval 12
ip ospf authentication message-digest
ip ospf message-digest-key 1 md5 hellokey
ip ospf cost 10

interface lo

access-list 100 permit 192.168.0.1
access-list 100 permit 192.168.0.2

router ospf
ospf router-id 192.168.1.100
log-adjacency-changes detail
redistribute connected
distribute-list 100 out connected
passive-interface bond0 //bond0 是用于登录并管理 DNS Server 的网络
network 192.168.1.0/24 area 0.0.0.0
network 192.168.2.0/24 area 0.0.0.0
! area 0.0.0.0 authentication message-digest

line vty

```

请注意，这里面涉及的主机名、地址、password、共享协商密钥等参数要根据实际情况进行替换。尤其是 hello-interval、dead-interval、度量值 Cost 和 ACL 等项目的取值仅供参考，实际配置时请与你的网络工程师进行充分沟通。如果服务器上的这些参数和网络设备上的不一致，将导致 OSPF 协商失败。

#### 4. 检查生效后的配置

完成所有配置后，可执行 ip route 命令检查是否生效。正常情况下，应能得到和下面相同的输出结果。

```

# ip route
192.168.1.0/24 dev em1 proto kernel scope link src 192.168.1.100
192.168.2.0/24 dev em2 proto kernel scope link src 192.168.2.100
192.168.0.0/24 dev bond0 proto kernel scope link src 192.168.0.100
169.254.0.0/16 dev em1 scope link metric 1004

```



```

169.254.0.0/16 dev em2 scope link metric 1005
169.254.0.0/16 dev bond0 scope link metric 1006
default proto zebra
    nexthop via 192.168.2.254 dev em2 weight 1
    nexthop via 192.168.1.254 dev em1 weight 1
default via 192.168.0.254 dev bond0

# ip route list table ospf1
192.168.1.0/24 dev em1 scope link src 192.168.1.100
192.168.2.0/24 dev em2 scope link
192.168.0.0/24 dev bond0 scope link
127.0.0.0/8 dev lo scope link
default via 192.168.1.254 dev em1

# ip route list table ospf2
192.168.1.0/24 dev em1 scope link
192.168.2.0/24 dev em2 scope link src 192.168.2.100
192.168.0.0/24 dev bond0 scope link
127.0.0.0/8 dev lo scope link
default via 192.168.2.254 dev em2

# ip rule
0: from all lookup local
32762: from 192.168.1.100 lookup ospf1
32763: from 192.168.2.100 lookup ospf2
32766: from all lookup main
32767: from all lookup default

```

### 8.3.4 如何守护 quagga 进程

quagga 的两个组件各行其道，ospf 进程负责运行 OSPF 协议，zebra 进程负责维护路由表条目。当停止 zebra 服务时，路由表中的相关条目会被修改。经测试得知，kill -9 zebra 不会触发切换，因为 zebra 被突然杀死时来不及维护路由表，虽然进程没有了，但路由表还是完好的。如果 ospfd 进程异常、设备断电、网络宕机都会出现切换。本地和横跨数据中心的切换时间基本都是秒级的，所以故障总时长和 OSPF 的 hello 包间隔时间相差不多。进程异常不会造成太大的影响，但是不希望手工维护 zebra 或者 ospf，你可以使用 monit 来守护着两个进程的状态。

monit 是一个提供监控服务的小工具。它承担 System V 服务的守护工作。monit 是一个非常轻量级的程序，不依赖任何第三插件、类库就可以胜任进程状态、文件系统变动等场景的监控工作，而且它还可以触发邮件通知和自定义动作等。

下面，我将给出一个 monit 守护 named、ospf 和 zebra 的配置示例。在目录 /etc/monit.d/ 下创建三个文件，并分别命名为 named、ospf 和 zebra。配置文件的详细内容如下所示。

/etc/monit.d/named 配置文件的示例：

```
set mailserver 192.168.0.73
```

```

set alert root@192.168.0.73
check process named with pidfile /var/run/named.pid
start program = "/etc/init.d/named start"
stop program = "/etc/init.d/named stop"
if failed port 53 protocol DNS then exec "ip addr del 192.168.0.1 dev lo:0;;ip
    addr del 192.168.0.2"
if cpu > 40% for 2 cycles then alert
if totalcpu > 60% for 2 cycles then alert

```

/etc/monit.d/ospfd 配置文件的示例：

```

#
set mailserver 192.168.0.73
set alert root@192.168.0.73
check process ospfd with pidfile /var/run/quagga/ospfd.pid
    start program = "/etc/init.d/ospfd start"
    stop program = "/etc/init.d/ospfd stop"

```

/etc/monit.d/zebra 配置文件的示例：

```

set mailserver 192.168.0.73
set alert root@192.168.0.73
check process zebra with pidfile /var/run/quagga/zebra.pid
    start program = "/etc/init.d/zebra start"
    stop program = "/etc/init.d/zebra stop"

```

配置完成后可以通过手工关闭或杀死被守护的进程进行测试。我们可以通过日志文件观察 monit 是如何工作的。

```

[CST May 13 21:11:35] error      : Aborting event
[CST May 13 21:12:35] error      : 'named' process is not running
[CST May 13 21:12:35] error      : Sendmail error: 501 5.1.3 Bad recipient address
syntax
[CST May 13 21:12:35] error      : Aborting event
[CST May 13 21:12:35] info       : 'named' trying to restart
[CST May 13 21:12:35] info       : 'named' start: /etc/init.d/named

```

另外，monit 还内置了一个嵌入式的 HTTP 服务，我们也可以使用图形的方式来管理监控这些进程。详情请参见图 8-7 和图 8-8。

### 8.3.5 BGP 在 Anycast 中的应用

前面我们只针对内网的可用性进行了讨论，现在来说说外部链路高可用的实现方式。外部链路，利用 BGP “透传” (pass-through) 是目前很多大型互联网可以考虑的一种方案。

BGP 是以自治域 (Autonomous System, AS) 为单位的。所谓“透传”，就是指数据能够从某一个 AS 中间穿越过去到达另外一个 AS。比如，这里有 A、B、C 三个 AS，它们之间的链接关系是  $A \leftrightarrow B \leftrightarrow C$ ，此时 B 就是一个具备“透传”能力的 AS，它允许 A 和 C 的数据从 B 中间通过。如果一个 AS 的外出路径只有一条，或者它只和一个其他 AS 相连接，



再或者它和所有的 AS 都是星形连接方式，那么对于这个 AS 来说就不具备“透传”的条件。

Home>

Running Monit on more than one server? Use [M/Monit](#) to manage all your Monit instances

monit 5.1.1

Monit Service Manager

Monit is **running** on station103.example.com with *uptime, 0m* and monitoring:

| System                                 | Status  | Load   |        |        | CPU                       | Memory            |
|--|---------|--------|--------|--------|---------------------------|-------------------|
| <a href="#">station103.example.com</a> | running | [0.03] | [0.05] | [0.03] | -1.0%us, -1.0%sy, -1.0%wa | 4.3% [2853084 kB] |

| Process               | Status  | Uptime | CPU  | Memory          |
|-----------------------|---------|--------|------|-----------------|
| <a href="#">named</a> | running | 11m    | 0.0% | 0.0% [30412 kB] |

Copyright © 2000-2010 [Tildeslash](#). All rights reserved. [Monit web site](#) | [Monit Wiki](#) | [M/Monit](#)

图 8-7 monit 监控页面

Home>named

Running Monit on more than one server? Use M/Monit to manage all your Monit instances

monit 5.1.1

Process status

| Parameter                           | Value   |
|-------------------------------------|---|
| Name                                | named   |
| Pid file                            | /var/run/named.pid  |
| Status                              | running   |
| Monitoring mode                     | active  |
| Monitoring status                   | monitored   |
| Start program                       | '/etc/init.d/named start' timeout 30 second(s)  |
| Stop program                        | '/etc/init.d/named stop' timeout 30 second(s)   |
| Check service                       | every 1 cycle   |
| Data collected                      | Sat May 13 21:47:51 2017  |
| Port Response time                  | 0.000s to localhost:53 [DNS via TCP]  |
| Process id                          | 12370   |
| Parent process id                   | 1   |
| Process uptime                      | 12m   |
| CPU usage                           | 0.0%  |
| Memory usage                        | 0.0% [31404kB]  |
| Children                            | 0   |
| Total CPU usage (incl. children)    | 0.0%  |
| Total memory usage (incl. children) | 0.0% [31404kB]  |
| Port                                | If failed localhost:53 [DNS via TCP] with timeout 5 seconds 1 times within 1 cycle(s) then restart else if succeeded 1 times within 1 cycle(s) then alert |
| Pid                                 | If changed 1 times within 1 cycle(s) then alert   |
| Ppid                                | If changed 1 times within 1 cycle(s) then alert   |
| Load average (5min)                 | If greater than 10.0 8 times within 8 cycle(s) then stop else if succeeded 1 times within 1 cycle(s) then alert   |
| Memory amount limit                 | If greater than 102400 5 times within 5 cycle(s) then stop else if succeeded 1 times within 1 cycle(s) then alert   |
| CPU usage limit (incl. children)    | If greater than 80.0% 5 times within 5 cycle(s) then restart else if succeeded 1 times within 1 cycle(s) then alert                                       |
| CPU usage limit (incl. children)    | If greater than 60.0% 2 times within 2 cycle(s) then alert else if succeeded 1 times within 1 cycle(s) then alert   |
| CPU usage limit                     | If greater than 40.0% 2 times within 2 cycle(s) then alert else if succeeded 1 times within 1 cycle(s) then alert   |

[Start service](#)[Stop service](#)[Restart service](#)[Disable monitoring](#)

Copyright © 2000-2010 Tildeslash. All rights reserved. [Monit web site](#) | [Monit Wiki](#) | [M/Monit](#)

图 8-8 进程实例页面

和 IP 地址一样，AS 需要一个 ID 来标识身份，IP 地址都会被标识属于哪一个 AS。联通的用户不可能使用电信的地址，所以，在寻址时路由必须要清楚源地址和目的地址分别所属的 AS 才行。同样，AS 的 ID 也分公有和私有两部分。64 512~65 535 这部分 ID 被预留给本地私有使用，对外出口使用的 ID 是你的运营商的。如果要想实现“透传”，就必须

购买一个公有的 ID。这样，你的网络就和各个运营商是平起平坐了，彼此之间可以实现数据“透传”。

“透传”的好处就在于链路的冗余。假设我有 A、B、C 三个数据中心，它们分别属于不同的运营商 X、Y、Z，那么我的链接模式是这样的： $A \leftrightarrow B \leftrightarrow C$ ， $A \leftrightarrow X$ ， $B \leftrightarrow Y$ ， $C \leftrightarrow Z$ 。如果运营商 X 到 A 的出口链路断了（也就是整个数据中心 A 的对外服务链路挂了），在没有“透传”的情况下，我们只能通过变更自己外网 DNS 的域名记录来引导 X 线路的用户访问其他的数据中心，但是因为 TTL 的影响，这个变更要想影响到用户是需要时间的。如果我们建立了“透传”链接关系，把本地所有数据中心当作一个 AS，并和 X、Y、Z 三家运营商之间构成网状关系，对外的出口都是一样的，那么不管是哪一个出口有问题，都可以通过其他的路由绕回来。这样就实现了内外全链路的高可用，路由流量切换的收敛速度是秒级的，比 DNS 要快得多。而且操作 DNS 切换和人员响应是需要时间的，即便维护人员实时在线并当即响应，至少也需要 5~10 分钟的时间。

那么，为什么要单独购买一个公网 ID，而不能蹭运营商的 ID 来实现呢？这是因为运营商的地址划分粒度不可能精细到刚好就是你所在的那一段。另外，你的地址段有可能是分散在很多个运营商或者 AS 当中的，如果每一个都做“透传”就太不划算了。“透传”的费用非常昂贵，这个服务是按流量收费的，流量太少的话，估计人家都不愿意跟你签。

什么时候需要购买一个 AS 的公网 ID 呢？评估一下，如果你的业务停止十分钟所带来的损失远远超过采购成本的时候就可以考虑了。

## 8.4 HTTP DNS

HTTP DNS 也是目前一个比较火热的技术，各大互联网公司都在相继开展对 HTTP DNS 构建的尝试。那么为什么放着好好的传统 DNS 不用，偏要使用 HTTP DNS 呢？

### 8.4.1 传统 DNS 的缺陷

目前国内传统 DNS 主要面临着三大缺陷。

第一，**域名缓存**。很多互联网接入运营商为了保证网内用户的访问质量和减少跨网结算，在内部搭建了内容缓存服务器，通过域名解析强行把资源指向内容缓存服务器，实现肥水不流外人田的目的。因为如果目标网站更新了域名解析，那么本地服务商如果没有及时维护，我们就有可能被严重误导。

第二，**解析转发**。有些本地服务商的 DNS Server 往往只配置了一个转发功能，而递归回来的地址有可能是错误的或者给出的 IP 地址和自己不是一个线路。比如，负责查询的 DNS Server 查到了一个电信的地址，可是用户的网络是联通的，这样就会造成访问缓慢或者无法响应。这种问题对于移动端的影响会更大。

第三，域名劫持。这和第一种情况差不多，域名缓存只是把流量扣在本地，而域名劫持就是赤裸裸的恶意攻击了。

### 8.4.2 HTTP DNS 的优势

HTTP DNS 的实现是绕过本地 DNS Server，通过在 URL 后面构造一个 DNS 查询请求，直接向目标站点的 DNS 进行查询。

从原理上来讲，HTTP DNS 只是更换了域名解析查询所使用的协议而已，但是这个小小的转变却带来了很多收益。

首先，它根治了域名解析异常，用户不会遭受劫持的威胁。

其次，它调度精准实时，可以保证将用户引导到访问速度最快的节点上。

再次，使用 HTTP 查询域名解析，可以根据业务逻辑对用户进行更细粒度的流量调度。而且它的改造成本低、扩展性强，有着非常广阔的应用空间。

### 8.4.3 HTTP DNS 长什么样

图 8-9 所示为一个 HTTP DNS 部署架构模型。四个不同的数据中心承载来自不同网络的用户访问。用户会根据网络线路决定使用哪个机房的服务。每个数据中心以负载均衡的模式挂载两台 HTTPDNS-SERVER 用于响应查询请求。HTTPDNS-SERVER 通过向 HTTPDNS-WEB 请求增量更新来实现域名解析的 up-to-date。两台 HTTPDNS-SERVER 的性能足以应付我们目前的访问量，即便是在出现负载不均衡的情况下，这种影响对于客户端来说基本上是无感知的。

如果当前数据中心的负载设备或者两台 HTTPDNS-SERVER 均出现异常，则该数据中心的 HTTP DNS 移动解析服务不可用。此时原本对应该线路的用户将向其他数据中心的负载设备去请求服务。例如，Mobile 数据中心出现异常时，移动用户的域名解析服务将产生延时效应。如果所有数据中心的负载设备或者 HTTPDNS-SERVER 全部异常，此时，用户会恢复到传统的 DNS 解析方式，向运营商的 LocalDNS 发起域名解析请求。此时 HTTP DNS 所带来的域名解析实时更新、域名防劫持等功能将失效。

同时，我们看到 HTTPDNS-WEB 是单点，如果它出现异常，将导致 HTTPDNS-SERVER 推送当前 QPS 消息失败。如果此时通过管控服务器对域名数据进行修改操作，则故障节点的数据将得不到更新。但 HTTPDNS-SERVER 已有数据的查询不受影响。所以，当 HTTPDNS-WEB 出现故障时，第一时间需要通知到管理员，在恢复 HTTPDNS-WEB 节点之前，必须即刻停止所有的变更维护操作。因为此时发布更新，故障节点所在数据中心的用户将无法得到最新的域名解析。如果你发布了一个促销活动，电信和联通的用户都可以参加，而移动用户却因为解析失败无法访问相关页面，这种差异对业务的影响是非常严重的。因此，当 HTTPDNS-WEB 发生异常时，应当立即停止执行任何数据变更的操作，先



尝试恢复故障节点。在某些特定情况下，有可能会出现急需域名变更的要求，比如此时又有其他生产服务器挂掉了，那么应当先停止故障 HTTPDNS-WEB 所在数据中心的 HTTP DNS 服务，让用户恢复到传统的 DNS 解析方式，然后再执行数据变更。这时候出现的数据不同步的情况，HTTPDNS-WEB 会在恢复正常后自动修复。

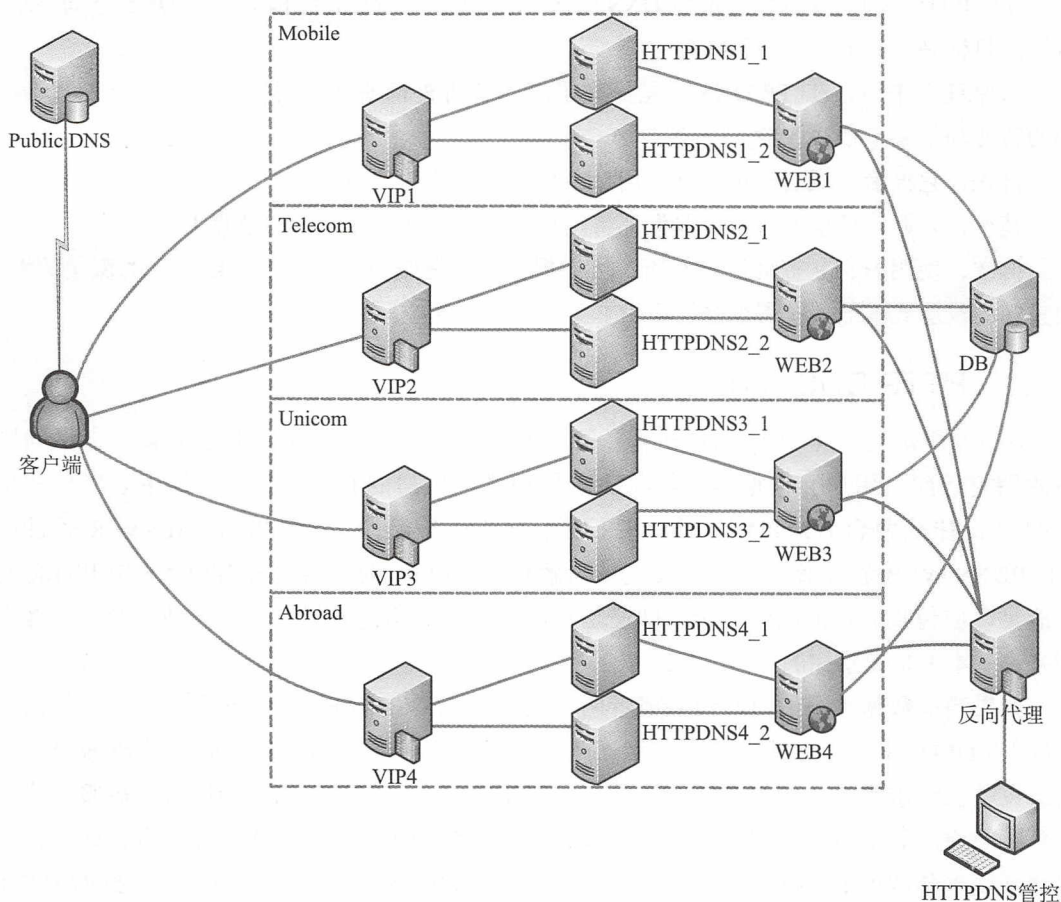
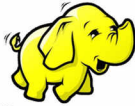


图 8-9 HTTP DNS 部署架构模型

#### 8.4.4 HTTP DNS 会取代传统的 DNS 吗

我只能说在相当长的一段时间内 HTTP DNS 是不可能取代传统的 DNS 的。因为 HTTP DNS 只适用于移动端的 APP，而无法取代传统 PC 端基于浏览器的域名查询。因为 HTTP DNS 在请求域名解析的时候需要 POST 一个 URL。它并不像传统的网络访问那么直观，只要在地址栏输入一个域名就好了。这个变化对于移动端的用户来说没有任何影响，因为 APP 在访问页面时都是通过点击按钮或者链接完成的。但它在 PC 端的场景却行不通，你不可能让用户自己在浏览器里面构造一个 URL。如果要在浏览器中实现，你得自己去开发



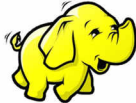
一个插件，可 PC 端的浏览器实在是太多了，要保证兼容性是一件非常困难的事情。除了防劫持和实时更新以外，HTTP DNS 的主要优势在于对解析速度的提升。但 PC 端的网络质量远远优于移动端，所以这个优势对 PC 用户而言没有特别的吸引力。再说，HTTP DNS 在没有全面普及的情况下，缺乏广泛的市场基础。HTTP DNS 是集中存储域名文件的，目前都是一些比较大的电商为了提升 APP 用户在移动端的体验而制定的，归根结底大家使用 HTTP DNS 的初衷还是为了自己，并非将目标指向公网基础设施的建设。所以，至少在目前这个阶段，HTTP DNS 无法替代传统 DNS，恐怕当前推动 HTTP DNS 发展的一切原动力也不是为了取代传统 DNS 而诞生的。

## 8.5 本章小结

本章讨论了 Anycast DNS 和 HTTP DNS 的原理和实现方式，通过实例为读者展示了如何构建 Anycast DNS 和 HTTP DNS。此外，就 BIND 当中的一些关键的核心知识点做了详细的讲解，以新老技术相结合的方式和读者一同分享了域名解析服务中的各种知识。

同时，笔者指出，Anycast DNS 是目前非常成熟的一种技术，完全可以部署在各种生产环境中。另外，尽管传统 DNS 面临诸多缺陷，但是短期内不会被 HTTP DNS 所取代。

下一章，我们将为大家讲述另外一种基础服务的构建——时间同步系统。



## 第9章

# 时间同步系统

《论语》云：“人而无信，不知其可也。大车无輹，小车无輹，其何以行之哉？”君子以信为本。人若是没有信用，则难以在社会上立足，而体现诚信最基本的一点就是守时。德国古典学者克里斯蒂安·蒙森说过：“不守时间就是没有道德。”

在我看来，守时不仅仅是一种素质和做人的态度，对于时间的准确把握更是关乎整个人类社会的兴亡。商海中时间就是金钱，战场上时间就是成败，手术台前时间就是生命。在今天，对于守时的严格要求不是什么道德绑架，而是人们不能容忍一两秒的时间偏差所带来的难以想象的灾难。

### 9.1 概述

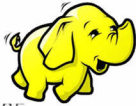
不论是军事、经济、科研还是日常生活，时间的准确性对于我们来说都是非常重要的。我们以证券交易所为例，假如在即将收盘时有一笔数额巨大的股票购买交易发生，但由于时间误差了一秒，导致这笔交易的失败。那么第二天再开盘时，股价有可能会发生比较大的变动，这将导致投资者重大的资金损失。因此，我们需要一个度量标准把时间维持在一个可以接受的精确范围之内。

时钟同步就是负责这件事儿的。时间同步又称“对钟”，即采用一个标准时钟作为参考，使其他时钟与标准钟对齐的过程。

#### 9.1.1 如何实现时间同步

实现时间同步需要三个要素——标准时钟、传输手段和实现协议。标准时钟来自于原子钟。原子吸收或释放能量时会发出一种电磁波，由于电磁波极其稳定，所以利用这个特性制造出来的原子钟的计时是非常准确的。有了度量标准，还需要将时间信号通过某种介质传播出去。早期人们利用无线电短波信号进行传输，但是由于短波的波长短，沿地表传输的地面波绕射能力差，传输距离短，而以天空波传输时，容易受到大气层中物质折射





的影响出现延迟,因此短波的时间精度比较差。后来人们又不断改进,使用超长波和长波进行信号传输。今天,利用卫星授时是实现全球范围的时钟精密同步的最好方式,卫星可在全球范围内使用超短波传播信号,大大提升了时钟同步的精度。说到实现协议,NTP (Network Time Protocol) 是目前最常用的、基于网络的计算机时间同步协议,它可以提供毫秒级精度的时间校正。

好了,现在三个要素都聚齐了,我们可以开始时间同步的工作了。在进行时间同步之前,我们需要选择一个时间源。时间源的种类有很多,包括GPS、北斗等卫星授时、内置原子钟授时和恒温晶振,等等。刚才我们说了,原子钟是标准时钟,卫星是传输介质,怎么这会儿全变成时间源了呢?所谓时间源,是指我们从哪里取得标准时间。原子钟所表示的时间是标准时钟不假,但是我们不能直接去拿。就好像米是国际标准长度,但你要测量长度就需要去买一把尺子。那我是买直尺、卷尺还是折尺呢?这就是选择时间源的问题了。

我们先说说恒温晶振。晶振是石英振荡器的简称,英文名叫作Crystal,它是时钟电路中最重要的一個部件。晶振的输出频率会随温度而变化,根据这个特性,晶振的类型分为温补晶振和恒温晶振两种,恒温晶振是目前频率稳定性和精确度最高的晶体振荡器。温补晶振的精度为 $10^{-7}$ ,恒温晶振的精度可以达到 $10^{-9}$ 。但是恒温晶振的开机性能不好,需要预热后才能进入工作状态。这个预热过程短则数分钟,长则需要一二十个小时。所以它不适合作为我们的首选时间源。

既然原子钟是标准时间,如果用它作为时间源肯定是最理想的了。不过高精度的原子钟大多用于军事领域,而民用领域的原子钟的精度和标准时钟之间还是有一定差距的。而且标准原子钟对于运行环境所需的条件也不是我们所能具备的。因此,内置原子钟更多情况下可以作为优于恒温晶振的备选守时时间源。

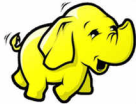
最常见的还是GPS、北斗等卫星系统授时的方式。下面我们就以GPS卫星系统授时为例,为大家介绍其中的原理。

### 9.1.2 GPS 卫星系统授时原理

GPS 全称为 NAVSTAR GPS (NAVigation Satellite Timing And Ranging Global Position System, 导航星测时与测距全球定位系统),它是一个由美国国防部开发的空基全天候导航系统,允许在地面或近地空间内获取一个通用参照系中的位置、速度和时间信息。

#### 1. GPS 如何授时的

GPS 系统包括卫星、地面监控系统和信号接收机三部分。GPS 卫星不断地向外发射自身的星历参数和时间信息,信号接收机在收到信号后,根据三角公式计算就能得到接收机自身所在的位置。我们日常的车载导航也是依靠这个方式来实现的。从1978年开始,美国国防部先后发射了总计24颗卫星,组成了GPS系统。这些卫星分布在6条卫星轨道之上,



任意时刻在水平线以上都会有 4~11 颗卫星存在。利用三颗卫星就可实现 2D 定位，四颗卫星则能够完成 3D 定位。所以说，GPS 定位的可靠性非常高。

说完了 GPS 定位，那么 GPS 上的时间信息又是从哪儿来的呢？每颗 GPS 卫星上也有自己的原子钟。GPS 卫星早期采用两部铯频标和两部铷频标，后来逐步变为采用更多的铷频标，使得它的精度可以达到世界协调时（UTC）的几纳秒以内。通常在任一时刻，每颗卫星上只有一台频标在工作。而 UTC 是由海军观象台的主钟来保持的，稳定性为若干  $10^{-13}$  秒。GPS 卫星系统的授时就借助发射无线电信号，把位置、位移和时间等信息传送给地面接收机，接收机通过计算得到自己的 3D 位置后，就能获取当地的 GPS 时间。

## 2. GPS 的时延和误差

GPS 卫星位于太空当中，当无线电信号不断地向地面发送时，大气电离层和对流层中的一些物质会让电磁波发生折射，从而使 GPS 信号的传输速度发生变化，导致接收机接收到信号的時刻要比卫星发送信号的時刻晚一些。就像电视节目中的现场报道一样，坐在演播室的主持人在向现场记者发出提问后，对方往往要等上一段时间才能回应。这就是时延现象。

除了时延的问题，GPS 在定位授时的过程中还要受到诸多因素的影响，导致出现授时误差。

第一个因素是来自于星钟上的误差。原子钟虽然非常精准，但仍会有一些微小的误差产生，而且卫星监控不是线性的，所以这些微小误差便会影响接收机在进行定位计算时的准确性。

第二个因素是星历误差。所谓星历，就是指天体运行随时间变化的精确位置或轨迹表，它是一个时间函数。在某一时刻计算 GPS 卫星位置所需的卫星轨道参数时，需要提供星历参与运算，但是计算出来的卫星位置 and 实际真实位置总会有一些差异，因此星历误差也称卫星轨道数据误差。

除此之外，还有很多其他影响因素。采用差分定位的方式可以抵消或者削弱这些影响因素。差分定位会根据两台以上接收机的观测数据来确定观测点之间的相对位置，它既可采用伪距观测量，也可采用相位观测量。差分定位比单点定位精确得多。所以，高精度的接收机通常采用双频或多频天线。

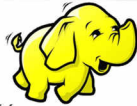
除了美国的 GPS 之外，全球卫星定位系统还有中国的北斗、欧盟的伽利略和俄罗斯的 GLONASS。

### 9.1.3 PTP

时间同步协议除了我们熟知的 NTP 以外，还有一种叫作 PTP。

PTP（Precision Time Protocol，精密时间协议）是一种用于网络时钟协议的解决方案。PTP 的时间精度远远优于 NTP，时间精度能够达到亚微秒级。





PTP 采用的是主从结构，主从之间的关系通过 BMC (Best Master Clock) 算法来确定。和 DNS 的主从关系不同，PTP 的 Master 和 Slave 角色是基于 Server 的端口而言的。如果端口是用来向下授时，那么它就称为 Master。反之，那些被动接收从上游返回的时间信息的端口就称为 Slave。但是对于一个 Master 端口而言，它所在的 Server 在向下授时的同时，也有可能要接收从上游设备下发下来的时间信息。因此，Server 根据端口数量又分为两种不同类型的 Clock。只有一个端口的 Server 称为 Ordinary Clock，而拥有多个端口的 Server 则称为 Boundary Clock。位于顶层能够直连 GPS 的 Server 称为 PTP Grandmaster，如图 9-1 所示。

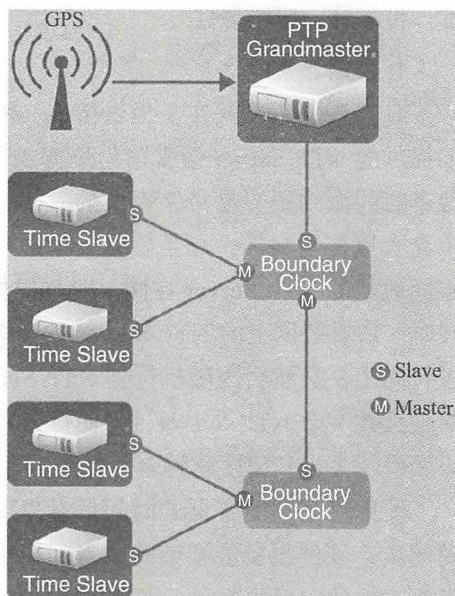


图 9-1 PTP Architecture

PTP 需要硬件支持，这一条件在很多网卡和交换机上都具备了。该硬件特性允许 PTP 记录传输延迟。PTP 之所以能提供更高的时间精度，完全是因为软件允许通过操作系统请求对 PTP 数据包的处理，让网卡驱动在 PTP 数据包收发的那个时刻追加一个时间戳的缘故。

我们举个例子。主从两端在收发信息时都将增加时间戳，例如 A 发送给 B，发送时在报文中加盖时间戳  $t_1$ ，B 在收到报文消息后，加盖接收时间戳  $t_2$ ，那么  $\Delta d_1 = t_1 - t_2$  就是发送一次数据包的延时时间，而一次完成的握手协商的总延时  $\Delta d = \Delta d_1 + \Delta d_2 + \dots + \Delta d_n$ 。假设网络是对称的，那么单向延时就是  $\Delta d/2$ ，主从时钟的 Offset 就是  $\Delta d_1 - \Delta d/2$ 。

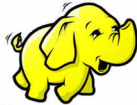
PTP 正是利用了时间戳标识才实现了时间同步的高精度。支持 PTP 的网卡拥有自己的“板载时钟”，它使用硬件时间戳来收发 PTP 消息，让自己和 PTP 主钟进行同步，之后再让系统时钟和自己完成同步。在有软件支持的情况下，可以直接用系统时钟对 PTP 消息加盖时间戳，让系统时钟直接与 PTP 主钟进行同步。

PTP 的硬件支持分为内核态和用户态两层。RHEL 通过网络驱动程序提供的模块，让内核实现对 PTP 的支持。使用 PTP 需要安装 linuxptp 软件包。linuxptp 包含 ptp4l 和 phc2sys 两个部分。其中，ptp4l 负责 Boundary Clock 和 Ordinary Clock，支持硬件时间戳和软件时间戳，硬件时间戳可用于“板载时钟”和 PTP 主钟的同步，软件时间戳可用于系统时钟和 PTP 主钟的同步。而 phc2sys 只有硬件时间戳，用于系统时钟和“板载时钟”的同步。

#### 9.1.4 为何要选用硬件时间源服务器

既然 NTP 协议能够实现软件时间的同步，而且也是免费的，为什么我们还要选择专业





的硬件时间源服务器呢？使用硬件时间源服务器主要是从以下三个方面来考虑的：授时精度、可靠性、可维护性。

### 1. 授时精度

时间源的意义就在于提供精准、恒定的时间信息。由于互联网的网络波动较大，公网上提供的 NTP Server 经常因为带宽或者速率问题被频繁地更替掉。而引入新的 NTP Server 参考时间源则很容易带来更大的时间偏差。就像一个与你合作了许久的伙伴突然被另外一个新人更替了一样，总是要有一个适应过程的。即便在网络相对稳定的情况下，多个来自不同地域的 NTP Server 在同一时刻提供的参考时间源也会因彼此之间的诸多差异影响终端授时。还有公网上的 NTP Server 的层级多半是 stratum 2 或者更高，如果让终端直接同它们进行同步，显然会因随机选择 NTP Server 而导致各个终端之间的时间偏差更大。如果内部单独设置一个 NTP Server，则使得 NTP Server 的层级变得更高。那就像翻录磁带一样，层级变得越高，其精度就越差。

硬件时间源服务器的时间信息来源于通过卫星或者原子钟，相对于通过互联网完成时间同步，硬件设备能够消除或者削弱由于网络延时所带来的种种不利影响，其授时精度要远高于软件实现的方案，能够更有效地进行实时校对。

### 2. 可靠性

时间源的正常运行依赖于软硬件的可靠度。从软件的角度而言，硬件时间源服务器采用了专业成熟的解决方案，基于嵌入式开发使其安全性、可靠性和性能都得到了充分的保障。在硬件方面，它剥离了传统服务器当中不必要的组件，使得硬件故障率大大降低了。另外，硬件设备还可以内置恒温晶振或原子钟等守时部件，以防止与外部时间源异常中断的情况发生。

### 3. 可维护性

硬件时间源服务器运行稳定，通常部署完毕后，很长一段时期内都无须人为干预。即便出现问题也会主动及时告警，维护工作量极少，可以说是非常省心。

## 9.1.5 如何选择硬件时间源服务器

选择一款硬件时间源服务器确实是一件比较头疼的事情，主要问题是设备参数。首先，一些专业参数不太容易理解。其次，不同厂商白皮书中所提供的参数并不统一，甚至连参考单位都不一致，非常容易混淆，难以实现横向比较。笔者当时参与时间源设备采购，为了弄清这些问题，也是大费周章。在这里，我将给大家分享一些自己的心得体会，谈一谈如何选择一款硬件时间源服务器。

### (1) 时间源

前面我们讲过，硬件时间源服务器的时间源主要分为卫星、原子钟、恒温晶振等几种。

关于卫星的选择, GPS 和北斗系统的授时精度都能够满足我们的需求, 而且两种卫星的可靠性都毋庸置疑, 二者任选其一即可。

因为天气或者磁场的影响, 卫星授时接收机会在某个极端时刻与卫星失联。另外, 数据中心可能由于信号不良、不具备施工条件或者政策法规等因素不适合安装卫星天线。因此, 通过内置原子钟来进行严格的守时工作就显得异常重要了。

综上所述, 如果数据中心部署条件优良, 且生产环境对于时间精确度的要求不是特别敏感, GPS 和北斗二选一就可以了。如果情况相反, 就一定要选用卫星 + 原子钟的模式。

### (2) 原子钟

目前市场上的原子钟产品总共分为三大类: 铷钟、铯钟和氢钟。铯钟和氢钟的时间精度高, 但价格十分昂贵, 往往用于国防或科研领域。民用领域则多采用铷钟, 铷钟具有短期稳定性高、体积小、便于携带、价格适中等特点。铷钟的主要参数有如下几个。

- 频率稳定度——指频率偏差的起伏程度;
- 漂移率——原子频标连续工作时频率随时间偏离的变化程度;
- 频率重现性——铷频标在开机和关机时相对平均频率偏差的一致性;
- 频率准确度——反映实际频率值和标称频率值的偏差最大范围。

值得注意的是, 漂移率这个数值没有统一度量值。有些厂商给出的是日漂移, 而有的厂商给出的是月漂移, 甚至是年漂移。由于原子钟主要是用来守时的, 所以漂移率是十分重要的一个参数。原子钟也会和卫星同步, 所以这个漂移率一定要问清楚是否是失去卫星锁定后的漂移率。有些厂商也把漂移率叫作守时精度。相对来说, 日漂移的可信程度还是更高一些的, 年漂移就显得有点儿不太靠谱了。因为这个度量的时间周期太长了, 参考意义也就不大了。而且年漂移这种数据基本上是无法测量的。原子钟的精度单位通常都是  $10^{-11}$  或者  $10^{-12}$ 。

### (3) PPS 精度

PPS 叫作每秒脉冲 (Pulse Per Second)。时间源服务器从 GPS 上获取了时间信息后, 如何输出表示呢? 它所依靠的就是这个 PPS 转换信号, 通过引入 PPS 信号的上升沿来标识 UTC 当中的整秒时刻。PPS 精度也决定了时间的表达精度。PPS 也包括锁定时和失去锁定时两种状态, 这两个参数通常都是以纳秒为单位的。失锁时的 PPS 精度大约比锁定时大十几倍, 这个数值是可以测量的, 可以要求厂商出具相关权威机构的检测报告。

### (4) NTP 测量精度

NTP 测量精度只出现在了部分厂商提交的检测报告中, 一般不高于几十微秒, 它指的是 NTP 观测时的偏离差值的精度。

### (5) 终端授时精度

终端授时精度是指向 client 下发的时间精度, 一般是  $1 \sim 10\text{ms}$ 。

### (6) 支持协议

选用的设备尽可能支持更多的协议，可以为将来使用和管理工作带来极大的便利。下列所列出的协议都是我们所推荐的。

- ☐ PTP v2 (IEEE 1588—2008);
- ☐ NTP v1, v2, v3 & v4 (RFC 1119 & RFC 1305);
- ☐ NTP Unicast, Broadcast, Multicast, Autokey TIME (RFC 868);
- ☐ SNTP (RFC 2030);
- ☐ MD5 Authentication (RFC 1321);
- ☐ RSA Encryption Algorithm;
- ☐ SNMP v1, v2 MIB II (RFC 1213);
- ☐ IPv4/IPv6 Hybrid;
- ☐ DAYTIME (RFC 867);
- ☐ DHCP (RFC 2131);
- ☐ Telnet (RFC 854);
- ☐ FTP (RFC 959);
- ☐ HTTP, HTTPS (RFC 2616);
- ☐ SSH, SCP, SFTP (Internet Draft);
- ☐ SYSLOG。

### (7) 监控告警

时间源服务器是一个比较重要的系统，它应当具备告警和监控的功能。这就要求时间源服务器能够支持对自身和下游客户端同步状态的监测，支持邮件和短信告警功能，最好还能够支持主流开源监控软件（如 Check\_MK、Nagios、Ganglia、Zabbix 等）的标准接口。

### (8) 其他参数

并发能力并不是一个特别重要的参数，通常情况下，我们会选用数台服务器作为二级参考时间源（Stratum 2）为终端服务器（Client）进行授时。大多数产品的并发能力至少都在 1000 以上。

平均故障间隔时间（Mean Time Between Failure, MTBF）是衡量一个产品可靠性的指标，单位是小时。这个数值越大，可靠性就越高。通常对于工控硬件设备的 MTBF 在 50 000~80 000 之间。

对于冗余的支持必不可少，网络链路至少要实现 Failover，提供双路电源，内置电容要确保在完全断电后至少提供 4 小时以上的时间供原子钟完成守时工作。

此外，还要关注设备对于工作环境的要求，架设天线还需要很多审批手续，所以在此之前都要事先协商好，以免在采购完设备后却发现无法部署或使用。



## 9.2 ntpd

ntpd 是传统的 NTP 服务实现的解决方案，它通过定期向时间源服务器发送请求来获取时间信息。这个定期请求并不是固定的时间间隔，ntpd 进程会在间隔期叠加一个随机的延迟值，这样做是为了防止网络风暴的产生。构建时间同步系统，通常会采用多层 NTP Server 的架构形式。硬件时间源作为一级授时，直接和从卫星获取时间。硬件时间源下联若干台服务器作为二级授时，为客户端提供时间同步服务。作为二级授时的服务器，内部部署的正是接下来我们要介绍的 ntpd。

### 9.2.1 ntpd 初始化

ntpd 服务在启动时，首先会读取 /etc/sysconfig/ntpd 文件进行初始化工作。这里面有几个重要的参数需要了解清楚。ntpd 的默认配置如下。

```
# Drop root to id 'ntp:ntp' by default.
OPTIONS="-u ntp:ntp -p /var/run/ntpd.pid -g"
```

参数说明如下：

- ❑ -u 用来指定由哪个账户和组来运行 ntpd 进程。为了安全起见，ntpd 服务在安装时单独建立了一个 ntp 账户用于启动 ntpd 服务。
- ❑ -p 用来指定标识 ntpd 进程 ID 的 pidfile。
- ❑ -g 表示在 ntpd 进程启动并进行时间同步时，ntpd 将忽略本地时钟和 NTP Server 时钟之间的时间偏差。不管两者之间相差得有多么离谱，ntpd 都会负责把时钟调整好。如果没有 -g 这个选项，当 offset 大于 1000s（16 分 40 秒）的时候，ntpd 就会放弃同步自动退出。当然，-g 只在 ntpd 第一次随系统启动时生效，如果中途重启 ntpd 服务，那么 -g 不再生效。一旦再次遇上 offset 大于 1000s 的情况，ntpd 依旧会自动退出。
- ❑ -x 用来尽可能地减少在时间同步中有可能发生的跃迁行为。关于时间同步方式这个问题这里暂且不表，我们会在 9.4 节当中详细说明。

计算机有两个时钟需要同步，一个是系统时钟，另外一个为硬件时钟，而 ntpd 默认只会同步系统时钟。Dell、IBM、HP 等一些国外的服务器，由于时区的默认设置问题，硬件时钟经常和系统时钟相差 8 小时。如果硬件时钟没有同步，当发生硬件故障时，硬件故障日志和系统日志的时间戳就不匹配，会给我们分析问题带来不小的麻烦。因此，需要在 /etc/sysconfig/ntpdate 文件中作如下修改。

```
SYNC_HWCLOCK=yes
```

如果需要手工调整两个时钟，则可以使用如下命令。

```
// 以系统时间为基准，修改硬件时间
```

```
# hwclock --systohcl-w
// 以硬件时间为基准，修改系统时间
# hwclock --hctosysl-s
```

不过，使用手工同步的方式会触发跃迁事件，类似的还有 `ntpdate`。这些命令请大家在执行的时候一定要小心，必须确保它们是在应用服务启动之前执行的，一旦应用服务上线，坚决不能触碰它们。

## 9.2.2 ntpd 配置文件

`ntpd` 的主配置文件是 `/etc/ntpd.conf`。下面，我挑选一些常见选项给大家介绍。

### (1) restrict

`restrict` 主要用来实现访问控制，例如下面这两个例子。

```
restrict 192.0.2.250
restrict 192.0.2.0 mask 255.255.255.0
```

另外，`restrict` 还支持如下选项。如果要使用这些选项，则需要将它们置于 `restrict` 命令的末尾：

- ☐ `ignore`——忽略所有数据包（包括 `ntpq` 和 `ntpd`）；
- ☐ `kod`——发送一个 Kiss-o'-death 数据包装死，用于减少那些恶意查询；
- ☐ `limited`——不去响应那些违反速率的查询或者使用已经弃用的命令（`ntpq` 和 `ntpd` 不受影响）；
- ☐ `nomodify`——不允许修改配置；
- ☐ `noquery`——不接受 `ntpq` 和 `ntpd` 查询，但不包括时间查询；
- ☐ `nopeer`——拒绝和其成为 `peer`；
- ☐ `noserve`——拒绝除 `ntpq` 和 `ntpd` 查询以外的任何数据包；
- ☐ `notrap`——拒绝 `ntpd` 控制消息协议的刺探；
- ☐ `notrust`——拒绝所有非加密认证的数据包。

### (2) peer

`peer` 表示对等关系，也就是说这台 Server 既接受来自对方的服务，同时也会为对方提供服务。

### (3) server

`server` 用于标识上游 NTP Server 的地址。

### (4) burst 和 iburst

我把 `burst` 和 `iburst` 称为扫射。正常情况下，我们一次只发送一个数据包，而这两个选项都会一次性发送八个数据包，就像用机关枪一样。不同的是，`burst` 是对当前一个可用的 NTP Server 进行扫射，由于 `burst` 发送了更多的数据包，在计算时间偏差量平均值的时候，

统计结果的数据质量会因为样本数量的增加而变得更好。所以, burst 可以用来提升统计数据的质量。但是请你千万不要乱用, 因为这会引发很高的负载。尤其是不能对着一个公网的 NTP Server 做这种事。而 iburst 的行为却恰好相反, 只有当一个 NTP Server 不可用时才会触发 iburst。使用 iburst 可以加快初始化的时间。使用 burst 和 iburst 时, 要将它们置于 peer 或者 server 命令的末尾。

#### (5) prefer

prefer 用于设定优先, 它表示在同等品质的 NTP Server 中优先使用这台 Server。使用 prefer 时, 要将它置于 peer 或者 server 命令的末尾。

#### (6) minpoll 和 maxpoll

这两个选项是用于修改默认更新间隔的。minpoll 和 maxpoll 的默认值分别是 6 和 10, 这两个数值代表  $2^6$  和  $2^{10}$ 。也就是说。默认情况下, 最小更新间隔是 64s, 最大更新间隔是 1024s。minpoll 和 maxpoll 的取值范围在 3~17 之间, 也就是更新范围可以从 8s 到 131 072s (36 小时 24 分 32 秒)。通常情况下, ntpd 需要不断地、周期性地和外部进行同步才能保持足够的时间精度。所以我们应当给 maxpoll 设置一个小些的值。

### 9.2.3 使用 ntpq 查询时间同步的状态

当 ntpd 服务启动后, 我们还可以使用命令 ntpq -p 来查询时间同步的状态。

```
[root@station103 ~]# ntpq -p
remote          refid          st t  when  poll reach  delay  offset jitter
=====
-61-216-153-107. 211.22.103.157 3 u   45m   1024  374   55.387   3.249   0.990
-61-216-153-106. 211.22.103.157 3 u   54m   1024  370   61.507   3.382  10.168
*dns1.synet.edu. 202.118.1.46   2 u   130   1024  377   12.847  -0.254   0.387
+jp2.ovear.info  10.84.87.146   2 u   208   1024  377   58.604  -5.879   2.511
+192.168.0.70    202.112.29.82 3 u   986   1024  377    0.600   0.032   0.439
```

接下来, 我们介绍一下这些字段的含义。

#### (1) remote

remote 指的是 server 或 peer, 而前面的标识代表状态字, 标明它们的状态。星号代表我们已经成功地与其进行了同步, 也就是我们正在“交往”的对象。加号代表这台 Server 是时间集群里面的“备胎”, 我们随时都有可能会使用它。减号则代表这台 Server 已经被我们所抛弃了。

除了上述三种状态以外, 其他状态字的具体含义参见表 9-1。

更多的字段含义可以参看 man ntp\_decode 手册。

#### (2) refid

本来, refid 最初的设计目的是为了服务器在异常情况下, 通过发送一个 KoD (kiss-o'-death) 数据包来通告用户的。它使用了一组非正式的 ASCII 字符串, 称之为 Kiss Codes。



不过，现在 refid 通常被当成用于标识自己的 Association ID，所以，现在这里使用 IP 地址来表明上游 NTP Server 的上家是谁。

表 9-1 状态字

| 代 码 | 消 息           | T     | 描 述              |
|-----|---------------|-------|------------------|
| 0   | sel_reject    | BLANK | 无效丢弃             |
| 1   | sel_falsetick | x     | 交集算法丢弃           |
| 2   | sel_excess    | ●     | 表溢出丢弃（未使用）       |
| 3   | sel_outlier   | -     | 集群算法丢弃           |
| 4   | sel_candidate | +     | 涵盖结合算法           |
| 5   | sel_backup    | #     | 备份               |
| 6   | sel_sys.peer  | *     | 系统对端             |
| 7   | sel_pps.peer  | ○     | PPS 对端（当优选对端有效时） |

我们在 station103 中看到，它有一台的上游服务器是 192.168.0.70，而 192.168.0.70 的上家是 202.112.29.82。我们到 192.168.0.70 这台服务器上，使用 ntpq -p 来查找一下。

```
[root@station70 ~]# ntpq -p
remote          refid              st t when poll reach delay offset jitter
=====
*dns1.synet.edu. 202.118.1.46      2 u 1012 1024 377  14.130 -0.897 0.694
+59.46.44.253    202.118.1.48      2 u  818 1024 377  22.578  3.206 2.705
+time6.aliyun.co 10.137.38.86      2 u  941 1024 377  13.979 -1.959 1.251
```

咦？怎么没有看到 202.112.29.82 呢？因为 remote 显示的是 DNS 反向解析的结果。为了看到服务器的 IP 地址，我们可以改用命令 ntpq -np 来查看。

```
[root@station70 ~]# ntpq -np
remote          refid              st t when poll reach delay offset jitter
=====
*202.112.29.82   202.118.1.46      2 u 1012 1024 377  14.130 -0.897 0.694
+59.46.44.253    202.118.1.48      2 u  818 1024 377  22.578  3.206 2.705
+115.28.122.198  10.137.38.86      2 u  941 1024 377  13.979 -1.959 1.251
```

原来 202.112.29.82 就是 dns1.synet.edu 这台服务器，而它就是 192.168.0.70 的上游 NTP Server。

(3) st

st 是指 stratum，也就是层级，代表你连接的 NTP Server 位于时间同步系统架构当中的第几层。

(4) t

t 是指类型，分为 u、b、l 三种。u 是 unicast，代表单播。b 是 broadcast，代表广播。l 是 local，代表本地。

### (5) when

我们发现 station103 里面总共有五台 NTP Server。其中, 前两台 NTP Server 的状态都是“减号”, 也就是被我们的 station103 所抛弃的。为什么会这样呢?

这时就要看 when 这个值了, 它代表我们和这台 NTP Server 最后一次同步成功后, 已经经过多长时间了, 默认单位是秒。我们看这两个 NTP Server 的 when 值, 它的后面有个字母 m, m 代表分钟。也就是说, 我们已经有很长时间没有从这两台 NTP Server 上成功地同步了。为什么这么长时间都同步不了呢? 很大程度上是因为我们已经好久没有联系了呢(例如网络问题)。既然大家不怎么联系了, 彼此之间疏于往来, 这样的 NTP Server 自然也就被我们所抛弃了。不过不用担心, 说不定过一段时间我们又会重新恢复关系的。

### (6) poll

poll 代表多久我们会和 NTP Server 联系一次, 这个数值就是通过 minpoll 和 maxpoll 来设定的。

### (7) reach

reach 代表成功更新的次数。注意这是一个八进制 (OCT) 的数值, 每成功更新一次, reach 都会增加, 但是它并不是简单地累加。下面, 我们将通过一个实例来理解它的算法。

首先, 我们把 reach 从八进制转换成二进制。当 reach 的值为 0 时, 转换成二进制后就是八个 0。每请求一次 reach 都会变化, 如果成功了就会增加 1。请注意, 这个 1 是二进制的 1。reach 首先左移一位, 然后末位加 1, 这样一来就变成了 00000001。如果第二次更新又成功了, 同理, 结果变成了 00000011。但如果第二次更新是失败的, reach 依旧会左移一位, 但不同的是, 这次是末位加 0, 结果就变成了 00000010。

如果我们看到 reach 的值为 377, 也就是二进制的 8 个 1, 它表明最近八次同步工作都是成功的, 说明这个时间源是稳定可靠的。一个时间源, 从 0~377 最少要历经 8 次 poll 的时间。而在这个过程中, reach 的变化无法直观地反应同步更新的成功与否。很多人误以为 reach 增加就代表成功, 这个说法大错特错。我们上面给出的例子就是一个典型的反例, 因为 00000010 > 00000001, 但是 00000010 却代表第二次同步更新是失败的。

接下来, 我们再举一个成功后 reach 值会减少的案例。请注意观看第一个 NTP Server (也就是 61-216-153-107) 的变化。

```
root@station103 ~]# date && ntpq -p
Fri May 19 10:05:09 CST 2017
remote          refid          st t  when  poll  reach  delay  offset  jitter
=====
+61-216-153-107. 211.22.103.158 3 u  1029 1024 343   54.794   2.185   1.285
-61-216-153-106. 211.22.103.158 3 u  1518 1024 302   152.719 -42.942  42.439
*dns1.synet.edu. 202.118.1.46    2 u   552 1024 377   13.119   0.630   0.661
-jp2.ovear.info  10.84.87.146    2 u   661 1024 377   59.704  -5.296   3.304
+192.168.0.70    202.112.29.82  3 u   334 1024 377    0.589   2.104   1.679
```

```
[root@station103 ~]# date && ntpq -p
Fri May 19 10:05:18 CST 2017
remote          refid          st t   when  poll  reach  delay  offset  jitter
=====
+61-216-153-107. 62.161.56.158  3 u     4   1024  307   54.891   0.855   2.116
-61-216-153-106. 211.22.103.158 3 u  1527  1024  302  152.719 -42.942  42.439
*dns1.synet.edu. 202.118.1.46   2 u   561  1024  377   13.119   0.630   0.661
-jp2.ovear.info  10.84.87.146   2 u   670  1024  377   59.704  -5.296   3.304
+192.168.0.70    202.112.29.82  3 u   345  1024  377    0.589   2.104   1.679
```

首先恭喜一下 61-216-153-107 同学，刚才我们之间还在冷战，那会儿它的状态还是减号，不是吗？不过现在这会儿，我们好像又重归于好了。

当时间到达 10:05:09 时，reach 的值是 343，when 的值是 1029，说明已经触发了更新的条件，因为 poll 的值是 1024。但是由于这条线路质量不太好，delay 的时间有些长，所以 when 的值还在一直增加。

当时间到达 10:05:18 时，我们看到 when 的值是 4，说明这次同步是成功的，上一次周期总共历经了 1034s。而此时 reach 的值是 307，我们用计算器算一下就知道发生了什么。

```
343 (8) = 11100011 (2)
307 (8) = 11000111 (2)
```

看到了么？reach 的值从 343 减小到 307，正是更新成功后左移一位且末位加 1 的结果。

为了让大家更清楚地了解这个算法，请参见表 9-2。它反映的是当同步过程中出现了一个同步失败事件时，reach 值随时间变化的完整过程。

表 9-2 reach 值

| Reach 字段值      | 解 释  |
|----------------|--|
| 377 = 11111111 | Time 0: Last eight responses from server were received |
| 376 = 11111110 | Time 1: Last NTP response was NOT received             |
| 375 = 11111101 | Time 2: Last NTP response was received                 |
| 373 = 11111011 | Time 3: Last NTP response was received                 |
| 367 = 11110111 | Time 4: Last NTP response was received                 |
| 357 = 11101111 | Time 5: Last NTP response was received                 |
| 337 = 11011111 | Time 6: Last NTP response was received                 |
| 277 = 10111111 | Time 7: Last NTP response was received                 |
| 177 = 01111111 | Time 8: Last NTP response was received                 |
| 377 = 11111111 | Time 9: Last NTP response was received                 |

#### (8) delay

delay 表示一个 NTP 数据包从发送到接收过程中所经历的时间。延迟会影响哪个时间同步，特别是当延迟是可变的时候。ntpd 需要估算延迟后再计算真实的时间。



### (9) offset

offset 就是时间偏差，它代表本地时钟和 NTP Server 时钟之间的差异，这也正是 ntpd 需要校准的时间量。

### (10) jitter

jitter (抖动) 是对网络延迟的标准差的测量。它表示两次时钟同步之间的时延的差值。参考它有助于补偿网络延迟。如果网络上的延迟是恒定的，则不会产生任何抖动。

我们可以简单地这样去理解 delay、offset 和 jitter 三者之间的关系：客户端原本和 NTP Server 之间的偏差是 offset，但由于 delay 引起了更大 offset。为了消除 delay 所带来的 offset，ntpd 采用了 jitter 补偿的方法。jitter 值越小，代表网络越稳定，其时间校正的准确度也就越高。

## 9.3 chronyd

除了 ntpd，NTP 还有另一种解决方案就是 chrony。chrony 是 RHEL 7 发行版的标配服务已经存在很多年了，chrony 由 chronyd 和 chronyc 两个程序组成。chronyd 是运行在系统后台的守护进程，chronyc 则用于监控和管理 chronyd。

### 9.3.1 chronyd 的优势

chronyd 之所以在新的发行版中取代了 ntpd，是因为 chronyd 具有如下 ntpd 所不具备的优势：

- ❑ chronyd 的守时效果更好，同步频率不需要像 ntpd 那么高；
- ❑ chronyd 因为守时效果好，使得它更加适应网络拥塞场景的出现；
- ❑ chronyd 通常可以更快地同步时钟并获取更好的时间精度；
- ❑ chronyd 能够迅速适应时钟速度的突然变化（例如晶振的温度变化），而 ntpd 则可能需要很长一段时间后才能安定下来；
- ❑ chronyd 默认从不会发生跃迁事件，这对于不熟悉 NTP 的用户可能是一个好消息，因为 ntpd 需要特殊配置后才可以防止跃迁事件；
- ❑ chronyd 可以在较大的范围内调整时钟速率（尤其是在虚拟机上）。

### 9.3.2 chronyd 配置文件

chronyd 的主配置文件是 /etc/chrony.conf。下面，我们挑选一些常见的选项来介绍。

#### (1) allow

allow 主要用来实现访问控制，例如下面这两个例子。

```
allow station1.example.com
allow 192.168.0.1/24
```

### (2) local

local 代表本地，下面这条语句用于防止服务器与本地自己进行同步。

```
local stratum 10
```

stratum 10 是一个很大的值，通常 NTP Server 层级要远远小于 10。这样做是为了防止 NTP Server 本地时间混入其中或者泄露到客户端（对客户端是可见）。

### (3) makestep

默认情况下，chronyd 是不会发生跃迁的，但如果时间偏差过大，就需要进行跃迁。此时需要使用 makestep 语句。下面这个示例的意思是，当 offset 大于 1000s 的时候进行跃迁，但仅限前 10 个时间更新。

```
makestep 1000 10
```

### (4) maxchange

maxchange 指定一个时钟更新所允许的时间偏差最大是多少。这个检查会在指定了允许多少次最大偏差之后才执行（在初始化阶段），如果发现超过这个数值，chronyd 会放弃更新并退出，同时产生 syslog 日志。下面这个示例的意思是，最大允许 1000s 的时间差异，每次都会检查，但会忽略前两次；如果把 2 设置成负数，则代表 chronyd 永久不会退出。

```
maxchange 1000 1 2
```

### (5) maxupdateskew

chronyd 除了调整时间，还会估算偏差界限，如果偏差过大，就说明这个参考不可信。maxupdateskew 参数是判断一个估算是否为不可靠的阈值。默认情况下，阈值为 1000 ppm。

```
maxupdateskew 10
```

典型的值是拨号网络是 100，局域网是 5 或者 10。为什么默认是 1000 呢？因为 chronyds 时刻都在对偏差做出评估，而且它的评估是加权组合的算法。如果一个稳定的度量方式和一个新出现的度量方式同时进入，总体有可能会产生一个比较大的偏差，当你没有独立的内部时间源的时候，更换 NTP Server 也是很正常的事情，因此要小心调整此数值。

## 9.3.3 使用 key 限制客户端访问

chronyd 也支持认证授权后的时间同步，以保证访问的安全性。使用 commankey 指令来启用一个密钥，commandkey 后面的 1 是一个 ID，这个 ID 必须是一个数字。

```
commandkey 1
```

/etc/chrony.conf 中指定了 key 文件的存放位置。

```
keyfile /etc/chrony.keys
```

在 chronyd 启动时候，会生成密钥文件 chrony.keys。

```
[root@station103 ~]# cat /etc/chrony.keys
#1 a_key
1 SHA1 HEX:E690B092F43E50FD7A930D85D6FE8A654DB5AB41
```

客户端需要在 `/etc/chrony.conf` 中作如下配置。

```
server x.x.x.x key 1
peer x.x.x.x key 1
```

### 9.3.4 跟踪时间同步过程

我们可以使用命令 `chronyc tracking` 来跟踪时间同步的过程。

```
# chronyc tracking
Reference ID      : 192.168.8.70 (ntp.example.com) // 上游
Stratum          : 2
Ref time (UTC)    : Wed May 24 05:58:09 2017
System time      : 0.000000482 seconds slow of NTP time
Last offset      : -0.000000002 seconds
RMS offset       : 0.000000608 seconds
Frequency        : 31.132 ppm slow
Residual freq    : +0.000 ppm
Skew             : 0.038 ppm
Root delay       : 0.000082 seconds
Root dispersion  : 0.010236 seconds
Update interval  : 3.9 seconds
Leap status      : Normal
```

参数如下：

- ❑ RMS offset: 代表长期以来一个 offset 的平均值。
- ❑ Frequency: 代表如果 `chronyd` 不参与同步系统将出现的偏差。
- ❑ Residual freq: 反映了参考时间源和当前 frequency 的差异，如果你的参考时间源获得了一个新的测量，它会加权取平均值；如果你的参考时间源的测量总是一致的，这个值会随着时间的推移慢慢趋近于零。否则它就会一直不会归零。
- ❑ Skew: 代表 Frequency 的误差边界。
- ❑ Root delay: 指从这里到 stratum 1 之间的时延。
- ❑ Leap status: 闰秒状态，包括 Normal、Insert second、Delete second 和 Not synchronized 四种。

### 9.3.5 检查时间同步状态

类似于 `ntpq -p`，`chronyd` 使用命令 `chronyc source` 来检查时间同步的情况。字段的含义大部分是类似的，我们只介绍那些不同。

```
[root@station103 ~]# chronyc sources
210 Number of sources = 3
```



| MS Name/IP address      | Stratum | Poll | Reach | LastRx | Last sample              |
|-------------------------|---------|------|-------|--------|--------------------------|
| ^* dns1.synnet.edu.cn   | 2       | 6    | 37    | 38     | -31us [-2255us] +/- 28ms |
| ^- ntp2.itcompliance.dk | 3       | 6    | 37    | 38     | +26ms [+26ms] +/- 200ms  |
| ^- biisoni.miuku.net    | 2       | 6    | 37    | 41     | +61ms [+61ms] +/- 144ms  |

1) MS。

□ M: “^”代表 server, “=”代表 peer, “#”代表时间源是通过本地连接的;

□ S: 状态字, 具体含义我们之前已经解释过了。

2) LastRx: 就是 ntpq 里面的 when。

3) Last sample: 指本地时钟和最后一次 NTP Server 的测量结果之间的偏移 (offset)。

括号里面是测量偏差, 括号左侧是调整偏差, 括号右侧是误差幅度。

命令 `chronyc sourcestats` 用来观察同步过程中对 drift rate 和 offset 的预估情况。

```
[root@station103 ~]# chronyc sourcestats
210 Number of sources = 3
Name/IP Address          NP  NR  Span Frequency Freq Skew Offset Std Dev
=====
dns1.synnet.edu.cn       12  7   523   -1.228    28.819   -27us  3267us
ntp2.itcompliance.dk     12 10   526  -13.676    25.104   +22ms  3132us
biisoni.miuku.net        12  6   525  -17.409    24.750   +49ms  3122us
```

4) NP: 当前保留的采样点数目, 通过线性回归可以估算出 drift rate 和 current offset。

5) NR: 最后一个线性回归得到的残余误差的样本个数, 这个数字太小会令线性回归的图形变成一条直线, 因而无法很好地作为参考。此时, `chronyd` 会丢弃旧的样本, 重新获取足够数量的样本个数。

6) Span: 指多长时间更新一次样本。

7) Frequency: 预估的残余频率。

8) Freq Skew: 预估的误差界限。

9) Offset: 预估的偏差。

10) Std dev: 预估样本的标准偏差。

## 9.4 如何处理闰秒

只要一提起时间同步, 就一定会牵引出一个不可避免的话题, 那就是闰秒。甚至可以这样讲, 如果没有闰秒, 时间同步会失去至少一半的话题, 我们可能也就不会那么关注它了, 这可是赤裸裸地“掉粉儿”啊。

### 9.4.1 闰秒是什么

为什么会有闰秒呢? 其实闰秒的出现和闰年是类似的。全世界有两种时间计量系统:

一个是基于地球自转的“世界时”(UT),还有一个是基于原子振荡周期的“原子时”(TAI)。“世界时”描述地球自转一圈是一天,但是地球自转不是绝对匀速的,所以不太可能恰好就是我们规定的 86 400s。不过,“原子时”的稳定性和精确性要比“世界时”高得多,大约是 100 万倍左右。也就是说,“原子时”和“世界时”之间存在着一定的偏差。如果放任不管,两者之间的差距会越拉越大。而闰秒的出现就是为了修复两者之间差距的。

既然如此,我们为什么还抱着世界时不放呢?直接使用原子时不就好了吗?之所以我们不能抛弃世界时是历史原因造成的。从人类起源到现在,我们大多数正常人都是遵循着“日出而作、日落而息”的生活习惯,我们的历法也全都是遵循世界时的。如果抛弃世界时而使用原子时,那么在经历过一段漫长的时间后,可能在某一天就会出现时间与昼夜交替严重不同步的情况。比如,你明明刚吃过中午饭没多久,太阳竟然就已经下山了。但是我们又不能直接去调整原子时,因为原子时是我们的度量标准。于是一个中间变量就出现了。

1972 年,出现了称为“协调世界时”(UTC)的折中时标。它以原子时的秒长为基础,在时刻上尽量接近世界时,确保它和世界时的偏差不超过 0.9s。因此,在适当的时候我们要为协调世界时增加或减少 1s。目前为止所有的闰秒事件都是增加操作,原因就是地球自转变慢的缘故,导致世界时比 UTC 慢。所以需要给 UTC 增加 1s,这样 UTC 就慢下来了,正好等着世界时追上来。

现在我们明白了一个道理:原子时是我们的度量标准,世界时是我们历法的基准,而基于原子时的协调世界时是用来和世界时进行对比的,当两者之间的偏差超过 1s 时,就会触发闰秒事件。

### 9.4.2 闰秒的危害

由于闰秒的调整是在 UTC 当中,而计算机又以 UTC 时间为基准,这样就产生了一个很大的问题。以增加闰秒来说,在整点时刻到来之前,UTC 会以 60s 或者两个 59s 的形式出现,这会让系统内核或应用程序因为无法识别而彻底傻掉。

在 Linux 早期的某个内核版本中,由于对闰秒标志位的处理不当,会导致 kernel 出现异常假死或者高负载的情况。除了系统内核以外,很多数据库或者应用程序依旧没有处理闰秒的能力。由于和闰年不同,闰秒是不可预测的。所以在代码上实现对闰秒的调整是有难度的。

据媒体报道,2015 年实施闰秒时,全球大约 2000 个计算机网络突然短暂中断,美国洲际交易所更是被迫中止交易长达 61min。而在此之前的 2012 年,全球数十家航空公司的计算机系统也因闰秒事件导致大面积停机。

为了解决闰秒,很多人建议干脆废除世界时就可以取消闰秒了。但是反对者则认为废除世界时会影响到历法。尽管从 1972 年到 2017 年,我们总共才发生了 27 次闰秒事件。

像我前面举的例子，可能需要经历几千年以后才会发生，但是反对者是基于“我们子孙万代还要统治地球几百万年”的角度考虑问题的。只要我们的历法还存在，就无法抛弃世界时。

所以到目前为止，闰秒事件的出现对运维团队来说仍是一个不容忽视的问题。

9.4.3 前辈们是怎么解决闰秒的

亚马逊的解决方案（见表 9-3）是把多出来的这 1s，在 24 小时的时间内消化掉。24 小时总共是 86 400s，那么如果每秒都延长 1/86 400s，也就是 0.000 011 574s。这么小的差距，不会对应用产生任何影响。

表 9-3 Amazon 的解决方案

| 日 期                        | UTC      | 系 统                | 偏 差 值          |
|----------------------------|----------|--------------------|----------------|
| June 30 <sup>th</sup> 2015 | 12:00:00 | 12:00:00           | +0             |
|                            | 12:00:01 | 12:00:00.999988425 | +1/86 400      |
|                            | ...      | ...                | ...            |
|                            | 23:59:60 | 23:59:59.5         | +1/2           |
| July 1 <sup>st</sup> 2015  | 00:00:00 | 00:00:00.499988425 | -43 199/86 400 |
|                            | ...      | ...                | ...            |
|                            | 11:59:59 | 11:59:59           | +0             |
|                            | 12:00:00 | 12:00:00           | +0             |

如表 9-3 所示，提前 12 个小时开始延时。当闰秒出现时，正常情况下系统时间应该刚好是 00:00:00，但由于延时策略，当前的系统时间是 23:59:59.5。在这 12 小时内，系统时间始终滞后于 UTC。但是由于闰秒的存在，UTC 相当于等了系统 1s，所以等 UTC 到达零点时，反而落后了系统时间 43 199/86 400s。之后的 43 199s，系统时钟继续通过延时策略运行，等着 UTC 追上来。当 11:59:59 时，两个时间终于一致了。

那么，追平时间为什么不是 12:00:00 呢？因为如果不是被延迟 1s，当前的系统时间应当是 12:00:00。但由于闰秒的缘故，时间已经减少了 1s，变成了 11:59:59。当时间来到 12 点时，系统时钟已经可以开始新的同步了。

9.4.4 晦涩难懂的术语

在讲述下面的问题之前，首先我们要把各种英文名词弄明白。不然到了后面，很多事情是解释不清的。

(1) step

step 直译就是脚步，在很多文章中，它还会和 backward 出现在一起，但并不是指倒退



的意思。backward 所谓的向后，是指时间轴上的向后，也就是我们常说的跃迁。

时间同步有两种方式。第一种方式就是用 step。如果本地时钟和 NTP Server 之间的时间偏差过大时，就需要使用 step 一下子把时间校正过来。你可以把 step 理解为瞬间移动。日常我们调整手表的时候，都是先将“表冠”拉出来让秒针停止走动，然后把时针和分针拨到当前时间后，再按下“表冠”让秒针恢复运动。秒针从停摆到再次恢复运动后，前后两个时间已经发生了瞬间的跳变，这就是所谓的跃迁。

## (2) slew

slew 是另一种时间同步的方式。这个词比较难以理解，slew 有很多种解释，其中有使之旋转到某一位置的意思，这是我认为最贴切的一种解释了。如果说得通俗一点儿，我觉得翻译成微调比较合适。

对于日常生活中的对钟，用 step 是非常实用的，因为我们只关心当前时间的准确性，而 step 可以在最短时间内迅速校正时间。但是对于应用程序来说，这样做却不可以。应用程序不但要关心当前时间，同时还要确保进程在整个生命周期内时间的连续性。我的进程一直运行得好好的，不可能上一秒还在 1 日的午夜 23 点，下一秒就跑到 2 日的中午 12 点了。尤其是数据库对于时间的连续性更是异常敏感。因此，通常只能使用 slew 的方式。

slew 是通过调整本地时钟的频率，进而改变单位 1s 的实际长度来实现对钟的。举个例子，假设本地时钟比 NTP Server 慢了 30s。slew 通过调整本地时钟频率，让自己的 1.0005s 变成现实时间的 1s。看上去本地时钟还是一秒一秒增加的，但每次走秒都会比实际要快那么一点点，直到追上 NTP Server 的时间。在这个同步过程中，总共需要 16 个多小时才能完成同步。

可以看出，slew 相比 step 要缓慢得多，但它的调整方式是最保险的。它的同步过程是“温水煮青蛙”的节奏，是在不知不觉中完成了对钟进行调整的任务，就像是用手把旋钮慢慢旋转过去一样。

## (3) smear

leap smear 是 Google 采用的一种处理闰秒的方式。在摄影领域，smear 叫作漏光或者弥散，如果你懂一点儿摄影技术，用弥散来解释 leap smear 基本上是可以领会到其中的含义的。简而言之，这就是一种补偿手段。当然，smear 只是 Google 的一种私有化命名，为了不会错意，我认为还是直接用 leap smear 最合适。

## (4) rate

rate 本来是速率的意思，但是我认为直译成速率是很难理解的。我在查询有关 slew rate 的词义时，在电学物理里面发现了电压转换速率（SR）这个词，其含义是在一个时间单位里电压升高的幅度，直观上讲就是方波电压由波谷升到波峰所需的时间，单位通常有 V/s、V/ms、V/μs 和 V/ns 四种。这样想来也许我们可以把 rate 理解成斜率。注意我是说理解，不是翻译。斜率在数学上有专有名词 slope。

我们知道斜率越大斜线越陡峭，反之斜线越平缓。当 rate 过大时，会导致系统的稳定性变差。比如，你可以想象一下汽车的急加速。

#### (5) offset

offset 我们并不陌生，通常计算机术语当中解释为偏移量，在这里翻译成偏差会更加自然一些。

#### (6) ppm

ppm 是 part per million 的缩写，也就是百万分之一。ppm 本身是没有什么含义的，和时间更是“八竿子打不着”。ppm 通常出现在频率描述当中，例如当 ppm 用作 frequency offset 时，它表示在一个特定中心频率下允许偏差的值，频率以赫兹作为单位。两者之间关系如下。

$$\Delta\text{freq}=(\text{freq}\times\text{ppm})/(10\text{E}+6)$$

$\Delta\text{freq}$  代表 Max frequency offset，而变化范围就取决于  $\text{freq}\pm\Delta\text{freq}$ 。

举个例子来说，如果标称 10MHz 的晶振所允许的频率偏差为 1ppm，那么这个频率偏差就是 10Hz，则频率变化应当在  $10\text{MHz}\pm 10\text{Hz}$  之间。

#### (7) drift

玩过赛车游戏的同学对 drift 肯定不会陌生，它有甩动、摆动的意思，就是我们常说的漂移、甩尾。在这里，drift 指的是钟摆幅度，因为座钟是通过改变钟摆幅度来调整走时速度的。在 /etc/ntp.conf 配置文件中的第一行就标明了 drift 文件的位置。

```
driftfile /var/lib/ntp/drift
```

driftfile 用来指定 drift 文件的存储位置。而 /var/lib/ntp/drift 文件中记录了本地时钟振荡器的调整频率，单位是 ppm。driftfile 指令和 ntpd -f 是等效的。如果指定的 drift 文件是存在的，那么 ntpd 服务在启动时会读取这个文件中数值，用于设置初始频率，这个频率并非一成不变，而是随着时间变化需要周期性地进行调整。之后每个小时（或更长时间），ntpd 会根据 offset、frequency、delay、jitter 等因素计算当前适合的频率。如果它的变化幅度大于 wander 的阈值，则会用新的频率去调整时钟，同时更新 drift 这个文件。更新的方式是：先写入一个临时文件，然后改名新文件去替换旧版。如果它的变化幅度小于 wander 的阈值，则不会修改 drift 这个文件。

### 9.4.5 怎么解决闰秒问题

要想解决闰秒问题，我们首先想一想闰秒是从哪里来的。闰秒是 UTC 中的概念，而且它是不可预知的。所以，对于闰秒标志位的发放来源，大多来自于上游 NTP Server 或者卫星授时系统，而守时的原子钟和系统本身并不会产生闰秒标志位。既然如此，在闰秒到来之前，断开外部时间源是首要工作，然后就是考虑如何去追平那一秒的问题。前面讲了

用 step 肯定是不行的，那就只有用 slew 了。我们可以参考亚马逊的经验来完成闰秒的处理工作。

### 1. 如何清除闰秒标志位

现在网络这么发达，闰秒变更这么大的事，一定会提前告知的。只要处理得当，通常下面的客户端都不会收到闰秒标志位。如果不幸真的收到了闰秒标志位，也不必过于紧张。客户端可以通过以下方法清除，然后假装没收到。

```
# service ntpd stop
# ntpdate -s 0 -f 0
# service ntpd start
```

-s 代表状态，后面的数值可以叠加，比如 3 就代表状态 1 和状态 2，具体的状态描述如下。如果你通过 ntpdate 看到有 INS 或者 DEL 的字样，就说明本地时钟收到闰秒通知了。

```
status 0x0 (),
status 0x1 (PLL),
status 0x2 (PPSFREQ),
status 0x4 (PPSTIME),
status 0x8 (FLL),
status 0x10 (INS),
status 0x20 (DEL),
```

### 2. 内核如何处理闰秒

早期的系统内核遇到闰秒时无法正确处理，要么直接 crash 掉，要么会产生非常高的负载，导致假死状态。新版本的内核采取 step 的方式来处理闰秒，在闰秒来临时，关闭系统时间一秒，然后再次打开，借此方式来跳过闰秒。不过，这种做法对应用来说是不负责任的。因为在关闭系统时间的那一秒内所做的任何事都无法解释。你打晕人家一秒后，再让人家醒过来，以为就没事儿了么？虽然我不知道那一秒发生了什么，但是人家的头还是很疼的哎。

### 3. ntpd 的 slew 方案

ntpd 默认情况是不会发生跃迁的，除非 offset 大于 128 ms。如果是因为网络拥塞尖峰所导致的大于 128 ms 的错误也会被忽略，但是这种情况一直持续了 900s，那么 step 也会被执行。

4.2.6 之后的 ntpd 支持了 slew 模式，它禁止内核自己去操作 step，并且忽略对闰秒的处理。当闰秒出现之后，再通过 slew 模式慢慢地将时间差修正回来。当在 /etc/sysconfig/ntpd 的参数中启用了 -x 选项时，128 ms 的默认阈值将被提升到 600s。所以 ntpd -x 并没有彻底断绝 step，只是大大地降低了 step 的可能性而已。为了处理 offset 大于 600s 的问题，还需要在 /etc/ntpd.conf 的主配置文件的第一行写入如下指令。

```
tinker panic 600
```



这条指令表明当 offset 大于 600s 时，ntpd 进程将退出不再参与同步工作。

但是这里面还存在一个问题。为了防止所有客户端同时并发所带来的网络风暴，同步校正的起始时间是在某个随机时段开始的，如果我们有多个 NTP Server，那么它们的时间修正的收敛曲线是不一致的。这个问题产生的影响对于多数据中心来说尤为突出。

#### 4. chronyd 的 slew 方案

RHEL 7 的新版本默认使用 chronyd 去替代 ntpd。chronyd 2.0 已经支持更加完善的解决方案了。

首先，调整 /etc/chrony.conf 中的 leapsecmode，这里总共有四个选项。

- ☐ system——让内核去处理（默认选项）；
- ☐ step——让 chronyd 替代内核去完成 step 工作，可以有效防止内核代码引发的 bug，类似于 4.2.6 之后版本的 ntpd；
- ☐ ignore——忽略闰秒，之后用常规手段修正时间，类似于 ntpd -x；
- ☐ slew——slewing 模式。

与 ntpd -x 不同，chronyd 在配置了 slew 模式后，它是正视闰秒的（如果有闰秒标志位），所以它依旧可以作为一个 NTP Server 存在。当它作为一个 NTP Server 存在时，slew 的整个过程是在本地完成的，而这个 slew 对于下游的客户端是不可见的，当闰秒触发时给客户端发送的同步方式是 step。

这里我们将 leapsecmode 修改为 slew。当然仅仅这样做还不够，到此为止这个配置和 ntpd -x 没什么太多的区别。chronyd 的优势在于，可以让多个 NTP Server 在闰秒调整全程中彼此之间尽可能保持步调一致。

接下来，需要配置 maxslewrate，让所有的 NTP Server 都使用相同的数值。该指令用于设定 chronyd 在 slew 时间段内所允许的最大速率，单位是 ppm。如果这个值设置得过大，会导致系统的稳定性变差，出现更多的 frequency error。限制住这个参数，就可以防止个别 NTP Server 的加速性能太好，出现抢跑的效应。

最后我们要说一下 leap smear。这个方案是谷歌在 2011 年提出的一种应用在多个 NTP Server 上的方法。它通过禁止下发闰秒标志位，然后靠 slew 来修正自己的时间。客户端并知晓闰秒的发生，它们也不用特别去配置，只要跟着 NTP Server 慢慢同步就是了。但是有一个前提条件，就是客户端只能和使用 leap smear 的 NTP Server 同步，不能再有其他方式的 NTP Server 掺和了。因为选举算法是过半当选，如果使用 leap smear 的 NTP Server 的数量小于使用其他方式的 NTP Server 的数量，leap smear 这个方式就会作废。如果两者数量相等，由于不知道该相信谁，同步就会停止。

Chronyd 2.0 已经支持了 leap smear。leap smear 的设计模型是这样的：当网络在一段时间内是与外界隔离的，客户端和多个使用 leap smear 的 NTP Server 保持同步，这个阶段客户端守时精度的要求不是特别严苛，但要求时钟步调尽可能保持一致。等到 NTP Server 长

时间离线后重新和外部同步时，会发送一个突然的改变，此时客户端会在一个随机时间后跟随，防止并发过高。那么当它们再一次保持一致时，中间需要经历一段时间。

我们可以这样来设置。

```
leapsecmode slew
maxslewrate 1000
smoothtime 400 0.001 [leaponly]
```

前两条就不必多说了，smoothtime 指令可以看作 maxslewrate 的进一步优化，让多个时钟步调尽可能地接近。它的语法如下。

```
smoothtime < Max_freq > < Max_wander > [leaponly]
```

Max\_freq 用来设置同步过程中所允许的最大频率偏差，单位是 ppm。其实关于最大频率偏差，我们刚才已经解释过了就是  $\Delta\text{freq}$ ，当然 freq 具体是多少我们无法测量，也没有必要去管，所以它是采用 ppm 来标识的。在 ntpd 里这个值最大可以设置为 500，只要不超过 500 就可以了，但实际的频率偏差没有那么大。

Max\_wander 用来设置频率偏差在每秒变化中所能改变的最大速率。maxslewrate 是针对整体，这里的 wander 是针对每秒的。这个值越小，同步的过程就越平滑，当然时间也就越长。

完成后，我们可以使用 chronyc -a smoothing 来观察同步情况。

```
# chronyc -a smoothing
200 OK
Active      : Yes
Offset      : -0.000003725 seconds
Frequency   : +0.117507 ppm
Wander      : +0.002000 ppm per second
Last update   : 3.9 seconds ago
Remaining time : 61.1 seconds
```

## 5. 总结

好，最后我们总结一下，如何处理闰秒问题。

首先要断开外部连接，不要让闰秒标志位下发，如果已经产生了闰秒标志位，可以使用 ntpd 清除。其次，绝大部分程序是不接受 step 的，因此不要让内核自己去处理。使用 ntpd -x 或者 leapsecmode slew 的方式来追平闰秒所带来的时间偏差。如果要确保多个时钟的同步步调，请选择 chronyd 的 leap smear 方式，而且所有的 NTP Server 都要启用 leap smear。

## 9.5 本章小结

本章讨论了时间同步的一些基础知识，并为读者阐述了如何选择硬件时间源服务器，

如何配置 ntpd 和 chronyd，以及如何处理闰秒。

在选择硬件时间源服务器时，笔者建议应当为对时间比较敏感的重要生产环境选用卫星授时系统 + 原子钟守时的组合方案。对于闰秒的处理，首先要断开外部连接，不要让闰秒标志位下发，并且使用 slew 的方式平滑过渡闰秒时刻所带来的时间偏差。

下一章我们将讨论配置管理工具在实际生产环境中的应用。



如果你问我当一个低调的土豪是什么感觉，我的回答是：每天坐着百公里 30L 油耗的百万豪车去公司上班，然后打开自己的笔记本电脑，靠在软椅里望着里面的几百个亿发一会儿呆。

从 2014 年年初入职，屈指算来已经第四个年头了。从当初的“十几个人、七八条枪”已经发展到了今天 30 多套存储、近万台服务器的规模。我像是一个见证人，目睹了整个平台发展壮大的全部历程。同时，我也是一个建设者，有时候在我看来它们就像是我的孩子，从呱呱落地的婴孩，到不断茁壮成长为一个英俊青年，每一个环节都倾注了我个人的心血与汗水。就拿这些服务器来说，每次执行维护任务时，感觉它们就像嗷嗷待哺的小鸟，每一个操作我都生怕漏掉了哪个节点，就像怕它们没有吃到食物一样可怜……

好了，我承认我已经抒情不下去了。总之，大家已经明白这章要讨论的主题是什么了，我们书归正传。

## 10.1 本章目的

学习有两种境界，一种是庖丁解牛的化境，另一种是化繁从简的炼境。任何一个配置管理工具都可以写上一本书。区区两三万字就想要全面剖析它们显然是不现实的。所以，我在动笔之前问了自己一个问题——本章的目的是什么？

显然，我不想写配置管理大全或是代码解析之类的东西，扪心自问自己也没有那个实力和深度。我在想，还有很多传统企业的运维团队正在技术转型的路上不断地探索，他们目前并不缺少帮助手册，也不缺少学习的热情和能力。只是他们还没有找通往新世界的“伟大航路”，尚未经历大规模节点应用的那个时代而已。

也许你听说过 Puppet，但不知道它究竟长什么样。也许你用过 SaltStack，但不知道它究竟有什么优势和劣势。明天领导要召开技术讨论会，大家就要用 Ansible 还是用 SaltStack 这个议题，最终要确定一个方案。我该说点儿什么？我们的选择最后会不会有坑？有没有

过来人给介绍点儿经验？如果站在这样一个技术选型的角度上看，我想本章还是有它的价值的。

因为本章实际操作的内容比较多，为了便于后续的描述，在正式介绍之前，我要先介绍一下我们的实验环境。本章参与实验的主机一共有四台，其中物理机 station103 担任配置管理主机的角色，redhat01、redhat02 和 redhat03 这三台虚拟机作为被管理主机节点。实验环境说明如表 10-1 和表 10-2 所示。

表 10-1 主机信息

| 角 色    | 主 机                    | IP 地址        |
|--------|------------------------|--------------|
| Admin  | station103.example.com | 192.168.0.73 |
| Client | redhat01.example.com   | 192.168.0.83 |
| Client | redhat02.example.com   | 192.168.0.85 |
| Client | redhat03.example.com   | 192.168.0.87 |

表 10-2 系统与软件版本

| 系统<br>版本 | CentOS | Parallel SSH | Ansible | Puppet | Salt Stack |
|----------|--------|--------------|---------|--------|------------|
| Version  | 6.5    | 2.3.1        | 1.9.2   | 2.7.26 | 2015.5.2   |

## 10.2 expect 与 Parallel SSH

20 世纪电子计算机的成本还非常昂贵。曾记得 1996 年，我的第一台个人电脑——一部搭载了 Intel Pentium® 75 的 PC 兼容机在当时的售价高达 15 000 多元。当时的服务器市场还是小型机的天下。一个负责 IBM 小型机的同事告诉我，那时干私活儿是最爽的，安装一台 AIX 至少也得几千块，而且这还是“友情出演”的优惠价。

尽管当时在 Unix 上已经有了 CFEngine 这样的系统配置管理工具。但是，在绝大多数场景下，执行命令比配置文件要频繁得多。管理员们还是更青睐使用 expect 和 pssh 来处理重复性的工作。

### 10.2.1 expect

expect 是我最早接触的批量执行工具。算起来，它竟然有近 30 年的历史了。expect 可以根据终端提示符模拟相应的标准输入，从而实现程序的自动交互功能，甚至你可以用它写一个简单的自动应答机器人程序。

expect 完全模拟了手工操作，只要你提前维护好逻辑判断，它几乎不会受到任何的环境限制。expect 是基于 Tcl 语言开发的。关于 Tcl，我了解的不是很多。除了 expect，另一

个我能联想到的恐怕就是 `mkpasswd` 了。它怪异的语法表达让我第一次学习 `expect` 的时候着实吃了不少的苦头，反复调试也令我丧失了对它深入研究的耐心。

虽然 `expect` 书写起来很痛苦，然而在实在没办法的时候，往往只有它能够救你一命。下面这段代码，我将它命名为 `expect_model.sh`。只要你的 SSH 服务可用，并且安装了软件包 `expect` 和 `openssh-clients`，就可以顺利地运行它。

```
#!/bin/bash

# expect 的函数主要用于实现生成一个符合 expect 语法的文件，后续用 expect 命令执行的所有操作都将
# 写在这个 exp 文件中
function f_expect ()
{
    cat > expect.exp <<EOF
    #!/usr/bin/expect

    # 禁用超时，等待响应 SSH 登录
    set timeout -1
    spawn ssh root@[lindex $argv 0]

    # 当出现 password 字样时，发送密码，如果需要确认 fingerprint 的提示，则发送 yes
    expect {
        "(yes/no)?" {
            send "yes\n"
            expect "*assword:" { send "$PASSWORD\n" }
        }
        "*assword*" {
            send "$PASSWORD\n"
        }
    }

    # 遇到命令提示符，发送命令
    expect {
        "#"
        {
            send "[lindex $argv 1]\n"
        }
    }

    # 命令执行完毕后，退出 expect
    expect "#"
    send "exit\r"

    expect eof
    EOF

    chmod 755 expect.exp
}

# 提供登录 SSH 的 root 密码
```



```

read -s -p "Please input root's password:" PASSWORD

#
echo >> expect.log
echo `date +%Y-%m-%d %H:%M:%S` >> expect.log
echo "-----" >> expect.log

f_expect

# 循环读取 IP 列表中的 IP 地址，然后根据地址依次 SSH 登录执行相关命令，并将结果返回。如果执行失败，
# 则将失败的 IP 记录到日志里面去
for IPADDR in `cat $1`
do
    ./expect.exp $IPADDR "$2"
    if [ $? -ne 0 ]
    then
        echo "$IPADDR is failed" >> expect.log
    fi
done

```

作为一个模板，这段 Shell 脚本没有太多的逻辑检查。它的使用方法如下：首先，在同一目录下创建一个主机列表文件，你要操作哪些主机，就将它们的 IP 地址或者主机名写到这里面。注意：每行只能写一个地址或者主机名，不要有空行或者其他无关的内容，例如像下面这样。

```

[root@station101 ~]# cat iplist
192.168.0.83
192.168.0.85
192.168.0.87

```

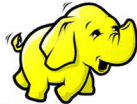
然后，开始执行脚本进行批量操作。执行时需要提供两个参数。第一个是主机列表的文件名，这里所指的就是 iplist。第二个是你要批量执行的命令，我们就以 hostname 为例来看它的执行效果如何。

```

[root@station101 ~]# sh expectmodel.sh iplist hostname
Please input root's password:spawn ssh root@192.168.0.83
root@192.168.0.83's password:
Last login: Fri Jun 23 10:19:39 2017 from 192.168.0.71
[root@redhat01 ~]# hostname
redhat01
[root@redhat01 ~]# exit
logout
Connection to 192.168.0.83 closed.
spawn ssh root@192.168.0.85
root@192.168.0.85's password:
Last login: Fri Jun 23 10:10:29 2017 from 192.168.0.71
[root@redhat02 ~]# hostname
redhat02
[root@redhat02 ~]# exit

```





```
logout
Connection to 192.168.0.85 closed.
spawn ssh root@192.168.0.87
root@192.168.0.87's password:
[root@redhat03 ~]# hostname
redhat03
[root@redhat03 ~]# exit
logout
Connection to 192.168.0.87 closed.
```

注意：如果命令有空格要用引号引起来，特殊符号则需要做转译。另外，这个脚本使用了默认端口 22 和 root 用户来登录，如果与你的实际环境不同，可以在 spawn 语句后面调整 SSH 的登录命令。

另外，脚本运行前要求用户提供 root 密码，该密码会被缓存起来，后面所有主机的 SSH 登录都会使用这个密码。如果执行过程中它和主机的 root 密码不匹配，你可以使用 Ctrl+C 组合键中断当前的连接，从而跳转到下一个主机，就像下面这样。

```
[root@station103 ~]# sh expectmodel.sh iplist "ls -l"
Please input root's password:spawn ssh root@192.168.0.83
root@192.168.0.83's password:
^Cspawn ssh root@192.168.0.85
root@192.168.0.85's password:
^Cspawn ssh root@192.168.0.87
root@192.168.0.87's password:
^Cspawn ssh root@192.168.0.89
root@192.168.0.89's password:
^C[root@station103 ~]#
```

脚本执行完毕后，会在当前目录下生成一个名为 expect.log 的日志文件，所有登录失败的主机信息都将被记录下来。你可以通过查看这个日志文件，来了解执行过程中哪些主机被漏掉了。

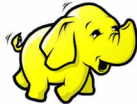
```
[root@station103 ~]# cat expect.log

2017-06-06 11:03:08
-----
192.168.0.89 is failed

2017-06-06 11:03:15
-----
192.168.0.83 is failed
192.168.0.85 is failed
192.168.0.87 is failed
192.168.0.89 is failed
```

## 10.2.2 Parallel SSH

Parallel SSH 的软件包叫作 pssh。顾名思义，它是用于并发执行 ssh 命令的一个工具。



pssh 包含了一系列相关工具：pssh、pscp、pnuke、prsync 和 pslurp。

和 expect 一样，首先我们要创建一个主机列表，每一行书写一个条目。其语法表达如下所示。

```
[user@]host[:port]
```

注意：如果不提供用户和端口号，pssh 默认使用 root 和 22 号端口进行 ssh 登录。相关的设置请和你的生产环境保持一致。

尝试执行如下命令，跟 expect\_model.sh 一样，我们在提示符下要输入用户名和密码，接着 pssh 输出了 id 命令的返回结果。

```
[root@station101 ~]# pssh -A -P -t 1 -h iplist 'id'
Warning: do not enter your password if anyone else has superuser
privileges or access to your account.
Password:
192.168.0.87: uid=0(root) gid=0(root) groups=0(root) context=unconfined_
u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[1] 10:39:51 [SUCCESS] 192.168.0.87
192.168.0.85: uid=0(root) gid=0(root) groups=0(root) context=unconfined_
u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[2] 10:39:51 [SUCCESS] 192.168.0.85
192.168.0.83: uid=0(root) gid=0(root) groups=0(root) context=unconfined_
u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[3] 10:39:51 [SUCCESS] 192.168.0.83
```

以下是 pssh 的常用选项：

- ☐ -P 是指允许打印命令执行后终端的输出结果；
- ☐ -A 是指要求输入密码；
- ☐ -h 后面跟上要执行的地址列表；
- ☐ -p 用于设置最大线程数；
- ☐ -t 用于设置超时时间；
- ☐ -x 用于设置额外的参数。

### 10.2.3 SSH 的通病

expect 和 Parallel SSH 都是基于 SSH 服务的，它也是我们日常维护工作时不可或缺的服务。SSH 服务稳定可靠，通信是加密的，安全性也很高。不过它也有很多不方便的地方。

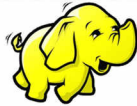
#### 1. sudo 的限制

很多实际生产环境中，不允许使用 root 用户直接登录 SSH。普通用户在执行命令时需要使用 sudo 来提权，而 sudo 默认不允许在非 TTY 上运行，当然这里的 TTY 也包括伪终端。这意味着你不能在后台去执行一个 sudo，否则系统就会返回下面这个经典的错误消息。

```
sudo: sorry, you must have a tty to run sudo.
```







解决方案是要注释掉 `/etc/sudoers` 文件中的这行。

```
# Default requiretty
```

但是,这种做法不太安全,不建议大家去修改系统的默认配置。`expect` 是模拟 SSH 登录的,它不会受到 `sudo` 的影响。`Parallel SSH` 可以使用参数 `-x '-t -t'` 来应对这个问题。选项 `-x` 用于设置额外的参数,它后面携带的不是 `pssh` 自己的参数,而是来自于 `ssh` 命令的参数。`'-t -t'` 这个参数所表达的含义是无视 `sudo` 配置文件并强制执行。

## 2. SSH 执行效率

`sshd_config` 的默认配置很容易引发登录时间过长的问題,这个问题的原因有两个。

第一个原因是 DNS 解析。由于 SSH 登录时要调用 DNS 完成名称解析,如果没有配置 DNS 服务或者 DNS 服务器上沒有相关的解析条目,服务器必须要等到 DNS 响应或者超时后才会进入下一个环节。优化 SSH 登录的首要任务就是解耦 DNS,不要被名称解析所绑架。

解决方法是在配置文件 `/etc/ssh/sshd_config` 中关闭 `UseDNS`。

```
UseDNS = no
```

第二个原因是无谓的验证尝试。SSH 登录支持很多种验证方式,例如 `Public-Key`、`GSS-API-with-XX`、`Password` 等。SSH 在登录时会尝试着将所有的验证方式按照优先级一一来过。如果第一个验证方式失败了,它会切换到下一个验证方式,直到所有的验证方式都失败才会退出程序。

最常见的验证方式是密码,但它的优先级最低,所以密码验证被排在最后一个。SSH 默认开启了 `GSSAPI` 认证,而 `GSSAPI` 是验证模式中最慢的一个。这也就是为什么很多使用密码登录 SSH 的用户抱怨它的密码输入提示框很久才出现的原因。

解决方法是在配置文件 `/etc/ssh/sshd_config` 中关闭 `GSSAPIAuthentication`。

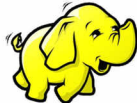
```
GSSAPIAuthentication no
```

即便如此,SSH 的登录速度还是不能令人满意。因为加密通信本来性能就不高,加之 SSH 协议又是无状态的,每次登录都要重新建立连接,注定了它的速度快不到哪里去。

不过,这个问题到了 `OpenSSH 5.6` 版本之后,有了一些小小的改进。新版本的 `OpenSSH` 支持 `ControlPersist` 方式,它可以固化保持登录状态,减少多次登录带来的效率问题。这对于在同一台主机上执行多次命令是一项有效的改善。

开启 `ControlPersist` 只需要在客户端一侧进行配置。在客户端用户的家目录下的 `.ssh` 目录中创建一个名为 `config` 的配置文件,并在其中添加如下内容即可。

```
# cat ~/.ssh/config
Host *
    Compression yes
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%r@%h-%p
```



```
ControlPersist 4h
```

### 3. 登录交互

SSH 默认在首次登录时会出现如下的提示。它要用户确认是否把 key 添加到 ~/.ssh/known\_hosts 文件当中。

```
Are you sure you want to continue connecting (yes/no)?
```

我们第一次执行批量管理操作时会受到这个交互提示的影响，我们并不想一次次地去输入 yes 来确认。为了避免这个麻烦，可以采取如下两种方法。

1) 在客户端配置文件 /etc/ssh/ssh\_config 中关闭 StrictHostKeyChecking。

```
StrictHostKeyChecking = no
```

2) 在命令行中使用选项 -o 携带参数。

```
# ssh -o StrictHostKeyChecking=no root@x.x.x.x
```

不过使用这种方式，在未经确认的情况下贸然去访问一台主机，存在一定的安全风险。最好还是采取公钥验证的方式来登录。因为密码验证方式的安全性不高，密码复杂度的设计和密码的存储分发都存在很多安全问题。

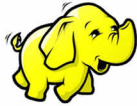
创建密钥的方式很简单，在管理端执行如下命令即可。

```
# ssh-keygen
# ssh-copy-id <USER>@<REMOTE_HOST>
```

### 4. 密码过期

严格地说，密码过期不是 SSH 的错。但既然 SSH 要通过登录来实现管理，就不得不面对这个问题。很多生产环境要求用户必须 90 天修改一次密码，当触发条件出现时，SSH 登录就会失败，即便是使用公钥验证依旧会受到这个问题的影响。使用选项 -v 打开 debug 日志信息可以看到，公钥验证是成功的，但是最后系统还是要求必须修改当前密码才行。

```
[root@station103 ~]# ssh -v 192.168.0.87
OpenSSH_5.3p1, OpenSSL 1.0.1e-fips 11 Feb 2013
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Applying options for *
debug1: Connecting to 192.168.0.87 [192.168.0.87] port 22.
debug1: Connection established.
debug1: permanently_set_uid: 0/0
debug1: identity file /root/.ssh/identity type -1
debug1: identity file /root/.ssh/identity-cert type -1
debug1: identity file /root/.ssh/id_rsa type 1
debug1: identity file /root/.ssh/id_rsa-cert type -1
debug1: identity file /root/.ssh/id_dsa type -1
debug1: identity file /root/.ssh/id_dsa-cert type -1
debug1: Remote protocol version 2.0, remote software version OpenSSH_5.3
debug1: match: OpenSSH_5.3 pat OpenSSH*
```



```
debug1: Enabling compatibility mode for protocol 2.0
debug1: Local version string SSH-2.0-OpenSSH_5.3
debug1: SSH2_MSG_KEXINIT sent
debug1: SSH2_MSG_KEXINIT received
debug1: kex: server->client aes128-ctr hmac-md5 none
debug1: kex: client->server aes128-ctr hmac-md5 none
debug1: SSH2_MSG_KEX_DH_GEX_REQUEST(1024<1024<8192) sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_GROUP
debug1: SSH2_MSG_KEX_DH_GEX_INIT sent
debug1: expecting SSH2_MSG_KEX_DH_GEX_REPLY
debug1: Host '192.168.0.87' is known and matches the RSA host key.
debug1: Found key in /root/.ssh/known_hosts:1
debug1: ssh_rsa_verify: signature correct
debug1: SSH2_MSG_NEWKEYS sent
debug1: expecting SSH2_MSG_NEWKEYS
debug1: SSH2_MSG_NEWKEYS received
debug1: SSH2_MSG_SERVICE_REQUEST sent
debug1: SSH2_MSG_SERVICE_ACCEPT received
debug1: Authentications that can continue: publickey,gssapi-keyex,gssapi-with-
mic,password
debug1: Next authentication method: gssapi-keyex
debug1: No valid Key exchange context
debug1: Next authentication method: gssapi-with-mic
Address 192.168.0.87 maps to localhost, but this does not map back to the address
- POSSIBLE BREAK-IN ATTEMPT!
debug1: Unspecified GSS failure. Minor code may provide more information
Credentials cache file '/tmp/krb5cc_0' not found

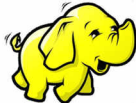
debug1: Unspecified GSS failure. Minor code may provide more information
Credentials cache file '/tmp/krb5cc_0' not found

debug1: Unspecified GSS failure. Minor code may provide more information

debug1: Unspecified GSS failure. Minor code may provide more information
Credentials cache file '/tmp/krb5cc_0' not found

debug1: Next authentication method: publickey
debug1: Trying private key: /root/.ssh/identity
debug1: Offering public key: /root/.ssh/id_rsa
debug1: Server accepts key: pkalg ssh-rsa blen 277
debug1: read PEM private key done: type RSA
debug1: Authentication succeeded (publickey).
debug1: channel 0: new [client-session]
debug1: Requesting no-more-sessions@openssh.com
debug1: Entering interactive session.
debug1: Sending environment.
debug1: Sending env LANG = en_US.UTF-8
You are required to change your password immediately (password aged)
Last login: Fri Jul 7 15:03:02 2017 from 192.168.0.73
Changing password for root.
```





```
(current) UNIX password:
```

正规的解决途径是老老实实地按照要求定期更新，但是这个操作必须是自动的。对于密码过期，你必须拥有一个额外的 channel 才能面对这种麻烦。快速恢复 SSH 通道的方法是：在新的 channel 上执行如下命令实现期限的更新。

```
# chage -d `date +%Y-%m-%d` < USER >
```

## 10.3 Ansible

Ansible 是近年来比较火爆的一款配置管理工具。它基于 Python 开发，集合了 Puppet、Cfengine、Chef、SaltStack 等众多配置管理工具的优点，能够实现系统配置、程序部署、命令执行的批量化操作。不过在实际的应用中，Ansible 还要面对很多现实的问题，例如，性能优化、节点的组织与添加等。下面，我就为大家一一道来。

### 10.3.1 创建 Host Inventory

Ansible 在安装完成后，默认会生成一个名为 `/etc/ansible/hosts` 的清单文件，Ansible 所有主机节点的组织管理工作都依靠这个清单文件来完成。清单文件使用方括号来定义一个组，这个组下面所有的对象都将成为该组的成员。成员可以是 IP 地址或者主机名，可以使用 `[X:Y]` 按照范围来定义。下面的示例文件展示了我们的实验环境是如何来组织节点的。

```
// 定义三台主机同属 ip 分组，redhat01.example.com 和 redhat02.example.com 为 web 分组成员，
redhat03.example.com 为 db 分组成员。
[web]
redhat0[1:2].example.com

[db]
redhat03.example.com

[ip]
192.168.0.83
192.168.0.85
192.168.0.87
```

### 10.3.2 如何自动添加节点

Puppet 和 SaltStack 这类配置管理工具都带有 Agent，它们的配置文件中都含有自动添加节点的相关配置。但是 Ansible 没有客户端，它是如何实现自动添加节点的呢？

首先，密码验证模式对于 Ansible 来说肯定不是一个好办法，所以第一要务就是把密码验证修改成公钥验证。其次，Ansible 第一次通过 SSH 管理一台主机的时候，需要将客户端的 Fingerprint 添加到 `known_hosts` 中，就像下面这样。

```
[root@station103 ~]# ansible all -m ping
paramiko: The authenticity of host 'redhat01.example.com' can't be established.
The ssh-rsa key fingerprint is 4a32f94c892d559b5cbf14fb193a33e8.
Are you sure you want to continue connecting (yes/no)?
```

一种方法是关闭 `ANSIBLE_HOST_KEY_CHECKING`，可以避免上述问题的出现，这和关闭 `StrictHostKeyChecking` 是类似的。

```
[root@station103 ~]# export ANSIBLE_HOST_KEY_CHECKING=False
```

另外一种更加合理的方法是使用 `ssh-keyscan` 指定添加主机节点的 key 到 `known_hosts` 中。主机节点可以从 `Host Inventory` 中直接获取。至于 `Host Inventory` 的更新和 `ssh-keyscan` 的执行，可以借助 `IaaS` 在系统部署完成后自动触发完成。

#### 1) 手工添加 key。

```
[root@station103 ~]# for i in 1 2 3
> do
> ssh-keyscan redhat0$i.example.com >> /root/.ssh/known_hosts
> done
# redhat01.example.com SSH-2.0-OpenSSH_5.3
# redhat02.example.com SSH-2.0-OpenSSH_5.3
# redhat03.example.com SSH-2.0-OpenSSH_5.3
```

#### 2) 扫描 Host Inventory 文件添加 key。

```
# ssh-keyscan -f `grep -v '\[' /etc/ansible/hosts`
```

### 10.3.3 组织主机节点

对于配置管理工具来说，在执行任何一项任务之前，首先要把即将执行操作的对象挑选出来，然后再统一发布执行命令。玩过即时战略游戏的同学对此一定最为熟悉不过了，在游戏里面我们管它叫做编队。

如何快速灵活、准确有效地完成编队操作是非常重要的。你肯定不能接受一个编队操作要花上好几分钟才能完成，慢吞吞的操作速度是无法适应激烈的战斗的。游戏前期的兵种很少，这时编队的动作很简单，只要全部圈起来就可以了。而后期会出现多个兵种组合，编队的操作就要适应灵活多变的局面，而且还要准确无误，不能选错了战斗单位。

DBA 或者 PE 使用配置管理工具的特点是全部操作基于某一项业务的变更需求。它们的模式就像单一兵种编队那样，只消用鼠标一圈就好了。除非业务增加或者删除节点，否则对内成员不会发生变动。

而 SE 则不同，系统级的配置管理是基于主机名或者 IP 地址的，每一次的操作对象都不固定。比如，SE 要为某几台主机打补丁，这些主机之间很可能没有任何关系，很难保证它们都来自同一个地方。

从这一点上，我们可以看出不同角色对于组织节点的要求是不同的。Ansible 支持很

多种编队方式。在介绍这些编队方式之前，我们先来看一个非常有用的选项，叫做 `--list-hosts`。它的作用是根据输入模式，在 Host Inventory 中找到所匹配的主机节点。在不熟悉 Ansible 之前，使用 `--list-hosts` 可以验证你的编队成员是否正是你想要操作的对象。

```
ansible <Pattern> --list-hosts
```

### 1. all

`all` 是 Ansible 中内嵌的一个关键字，它代表 Host Inventory 中所有列出的主机节点。通过下面的例子我们看到，Ansible 的选项还支持缩写。

```
[root@station103 ~]# ansible all --lis
redhat01.example.com
redhat02.example.com
redhat03.example.com
192.168.0.83
192.168.0.85
192.168.0.87
```

### 2. 按组编队

按组编队是典型的业务操作模式，也是 DBA 和 PE 最常用的方法。将主机按照业务划分成不同的组，操作时按组对某一项业务进行统一的配置管理。

```
// 依照 web 分组列出对应的主机有哪些
[root@station103 ~]# ansible web --lis
redhat01.example.com
redhat02.example.com
// 依照 db 分组列出对应的主机有哪些
[root@station103 ~]# ansible db --lis
redhat03.example.com
```

### 3. 根据 IP 地址编队

根据 IP 地址编队是一个非常灵活的操作模式，由于 IP 是数值，Ansible 支持地址范围的指定方式。

```
// 依照 IP 地址，列出 192.168.0.83 到 192.168.0.85 地址段所对应的主机有哪些
[root@station103 ~]# ansible '192.168.0.83:192.168.0.85' --lis
192.168.0.83
192.168.0.85
```

### 4. 按照通配符或者正则表达式组织

比写一个具体的 IP 地址更便捷的方法是使用通配符和正则表达式。对于 SE 来说，这个模式是非常重要的。注意：如果要使用正则表达式，需要在前面添加“`~`”。

```
// 依照 IP 地址，列出 192.168.0.80 到 192.168.0.89 地址段所对应的主机有哪些
[root@station103 ~]# ansible '192.168.0.8?' --lis
192.168.0.83
192.168.0.85
```



```

192.168.0.87
// 依照 IP 地址, 列出结尾含 3 的地址段所对应的主机有哪些
[root@station103 ~]# ansible '*3' --lis
192.168.0.83
// 依照 IP 地址, 列出 192.168.0.83 到 192.168.0.85 地址段所对应的主机有哪些
[root@station103 ~]# ansible '~192.168.0.8[3-5]' --lis
192.168.0.83
192.168.0.85

```

### 10.3.4 Ad-Hoc

Ad-Hoc 被称为 Ansible 的命令集, 它采用模块调用的方式, 可以迅速地完成一系列紧急的临时性任务。Ansible 大约支持几十种模块应用, 不过我们没有必要全部掌握, 只需熟练使用如下这些模块就够了。

#### 1. ping 模块

ping 模块不是简单地 ping 一个 IP 地址, 而是用来测试 Ansible 和被管理主机之间的 SSH 连通性的。

```
# ansible all -m ping
```

#### 2. shell 模块

shell 应该是 Ad-Hoc 中最有价值的一个模块了。shell 模块让你觉得就像在终端上执行命令一样。你平时怎么用, 在这里都是一样的。有了它, 你甚至不必学习其他模块的使用方法, 一样可以很好地完成批量操作的任务。

```
ansible all -m shell -a "<Command>"
```

#### 3. yum 模块

顾名思义, yum 模块用于软件包的管理。如果需要同时管理多个软件, 可以使用逗号进行分隔。

```

# ansible all -m yum -a "name=sysbench state=present"
# ansible all -m yum -a "name=telnet state=absent"
# ansible all -m yum -a "name=httpd state=latest"

```

#### 4. service 模块

service 模块使用起来比较方便, 它允许在一行里同时设置服务状态和启动级别, 并将结果返回。它对于差异化服务管理 (如 CentOS 7 和 CentOS 6) 非常有帮助。

```

# ansible all -m service -a "name=ntpd state=started enabled=yes runlevel=3"
# ansible all -m service -a "name=postfix state=stoped enabled=no"
# ansible all -m service -a "name=httpd state=restarted"

```

#### 5. 返回值

需要注意的是, Ansible 的返回结果和返回值指的是被管理主机上的系统返回值。Fai-

led 和非 0 并不完全代表是否执行成功，这一点和后面我们要讲到的 SaltStack 是有所不同的。我们看下面这个例子，所谓的失败只是 grep 没有找到匹配对象而已。

搜索客户端中包含 an 的安装包。

```
[root@station103 ~]# ansible ip -a 'rpm -qa |grep an'
192.168.0.87 | success | rc=0 >>
libsemanage-2.0.43-4.2.el6.x86_64
slang-2.2.1-1.el6.x86_64
cronie-anacron-1.4.4-12.el6.x86_64

192.168.0.85 | success | rc=0 >>
libsemanage-2.0.43-4.2.el6.x86_64
slang-2.2.1-1.el6.x86_64
cronie-anacron-1.4.4-12.el6.x86_64

192.168.0.83 | success | rc=0 >>
libsemanage-2.0.43-4.2.el6.x86_64
slang-2.2.1-1.el6.x86_64
cronie-anacron-1.4.4-12.el6.x86_64
```

我们把约束条件变得更严格一些后发现没有找到任何安装包。请注意每个主机的返回结果下面多了一行空行，代表什么也没有找到。

```
[root@station103 ~]# ansible ip -a 'rpm -qa |grep ans'
192.168.0.87 | FAILED | rc=1 >>

192.168.0.83 | FAILED | rc=1 >>

192.168.0.85 | FAILED | rc=1 >>
```

## 6. 引号和转译符

因为执行命令外面包含了一对引号，所以如果我们的命令本身也有引号，为了不引起歧义，那么就必须把里外两对引号错开使用。另外，如果命令中有特殊符号要特别注意。以“\$”为例来说，由于整个命令作为 Ansible 的一个参数被传递出去，此时的“\$”就是变量取值的含义。所以，如果你要想使用“\$”作为一个普通的字符，必须在前面增加转译符“\”。不要幻想使用单引号，因为作为参数的单引号已经失去了系统原有的强制转换字符串的功能。

我们来看两个例子，这里使用 usermod 修改用户 luci 的密码，修改完成后，我们发现散列中的部分字符串丢失了。这是因为“\$”后面的字符串被当成了变量，而这些并不存在的变量取值为空。

### 1) 使用 usermod 修改密码。

```
[root@station103 ~]# ansible db -m shell -a 'usermod -p "$6$IXXP34pU$1YZDgmKUV/ccSq
IIBfQYi3mLQGFBP8lRRpVLTkXtIaUrQQYhcs01rUJ87oLjQ/gLx25AdnJkbBQlg3UvEjOp." luci'
redhat03.example.com | success | rc=0 >>
```

## 2) 密码丢失部分字符串。

```
[root@station103 ~]# ansible db -m shell -a 'grep luci /etc/shadow'
redhat03.example.com | success | rc=0 >>
luci:YZDgmKUV/ccSqIIBfQYi3mLQGFBP8lRRpVLTlkXtIaUrQQYhcs01rUJ87oLjQ/gLx25AdnJkbBQ
lg3UvEjOp.:17324:0:99999:7:::
```

## 3) 再次使用转译符修改。

```
[root@station103 ~]# ansible db -m shell -a 'usermod -p "\$6\$IXXP34pU\$1YZDgmKUV/ccS
qIIBfQYi3mLQGFBP8lRRpVLTlkXtIaUrQQYhcs01rUJ87oLjQ/gLx25AdnJkbBQlg3UvEjOp." luci'
redhat03.example.com | success | rc=0 >>
```

## 4) 此时再看才是正常的。

```
[root@station103 ~]# ansible db -m shell -a 'grep luci /etc/shadow'
redhat03.example.com | success | rc=0 >>
luci:$6$IXXP34pU\$1YZDgmKUV/ccSqIIBfQYi3mLQGFBP8lRRpVLTlkXtIaUrQQYhcs01rUJ87oLjQ/
gLx25AdnJkbBQlg3UvEjOp.:17324:0:99999:7:::
```

同理，下面是一个 `awk` 的例子。如果第二条命令中没有添加转译符，那么 `$NF` 的值为空，其返回结果应当和第一条命令是一样的。

```
[root@station103 ~]# ansible ip -m shell -a "sysbench --num-threads=4 --max-requests=100
--test=cpu --cpu-max-prime=100 run|awk '/total time:/ {print}' "|grep -v SUCCESS
total time: 0.0002s
total time: 0.0002s
total time: 0.0002s"
```

```
[root@station103 ~]# ansible ip -m shell -a "sysbench --num-threads=4 --max-
requests=100 --test=cpu --cpu-max-prime=100 run|awk '/total time:/ {print
\$NF}' "|grep -v SUCCESS
0.0002s
0.0002s
0.0002s"
```

## 7. sudo

前面我们提过 `sudo` 后台执行的问题。为此，Ansible 自带 `-s` 选项来应对这个问题，这一点类似于 Parallel SSH 的 `-x -t -t`。

```
[root@station103 ~]# ansible db -u se -m shell -a 'grep root /etc/shadow'
redhat03.example.com | FAILED | rc=2 >>
grep: /etc/shadow: Permission denied
```

```
[root@station103 ~]# ansible db -u se -m shell -a 'sudo grep root /etc/shadow'
redhat03.example.com | FAILED | rc=1 >>
sudo: sorry, you must have a tty to run sudo
```

```
[root@station103 ~]# ansible db -u se -s -m shell -a 'grep root /etc/shadow'
redhat03.example.com | success | rc=0 >>
root:$6$0ru2IgYjDo.Pmea9$AUT7nVdBur1HtZVDmySGyRbxLZtgTabFQVNUuSnWQgy6gLObjxLe2mp
```



```
DlAcBr6G5WK7Gc98RdFMizVeL9ImA8.:17226:0:99999:7:::
```

## 8. 简化输入参数

上面那个 `sudo` 的例子在执行的时候，在命令的后面要带一大堆参数是不是很烦人？Ansible 默认的账户和模块分别是 `root` 和 `command`，你可以通过修改配置文件 `/etc/ansible/ansible.cfg` 实现输入简化。

```
remote_user = se
module_name = shell
```

Ansible 默认的 `command` 模块和 `shell` 模块之间有什么区别呢？顾名思义，`command` 模块只能单独执行一条命令，它不支持管道等特殊的操作，无法实现命令的组合。

我们看下面这个例子，管道被当成一个普通字符来用，导致 `ls` 命令尝试着去寻找一个名为 `/tmp|grep` 的文件。

```
[root@station103 ~]# ansible '192.168.0.83' -m command -a 'ls /tmp'
192.168.0.83 | success | rc=0 >>
tmpDuonSh
tmpPYkY4w
tmpvRuhBU
tmpy0Dtf2
yum.log

[root@station103 ~]# ansible '192.168.0.83' -m command -a 'ls /tmp|grep yum.log'
192.168.0.83 | FAILED | rc=2 >>
ls: cannot access /tmp|grep: No such file or directory
ls: cannot access yum.log: No such file or directory

[root@station103 ~]# ansible '192.168.0.83' -m shell -a 'ls /tmp|grep yum.log'
192.168.0.83 | success | rc=0 >>
yum.log
```

不仅仅是管道，连接符也是如此。

```
[root@station103 ~]# ansible '192.168.0.83' -m shell -a 'ls /tmp/yum.log && echo yes'
192.168.0.83 | success | rc=0 >>
/tmp/yum.log
yes

[root@station103 ~]# ansible '192.168.0.83' -m command -a 'ls /tmp/yum.log && echo yes'
192.168.0.83 | FAILED | rc=2 >>
/tmp/yum.logls: cannot access &&: No such file or directory
ls: cannot access echo: No such file or directory
ls: cannot access yes: No such file or directory
```

## 9. 输出乱序

在组织主机节点的时候，我们就已经看到一个现象。每一次执行的返回结果的输出顺序

都不固定。Ansible 是按照 Inventory 文件里面节点的排列顺序来执行的，而输出的返回结果却没有按照顺序排列。这是由于并发执行，谁先执行完毕终端上就先返回谁的执行结果。

如果执行的是修改类的命令，我们是不太在意顺序的。但是换成了读取类的命令情况就不一样了。比如，一个存储下挂载了十台主机，我想去检查它们的 IP 地址、主机名、多路径和磁盘挂载等诸多情况。我会希望每次结果能够按顺序输出。由于读取类的命令的目的就是查看结果，所以操作的节点数量不会很多，可以临时采用单进程的执行模式来实现。

```
# ansible all -f 1 -a 'ls /tmp'
```

### 10.3.5 Playbook

Playbook 是 Ansible 的核心部分。可以这样讲，如果你没有使用过 Playbook，那 Ansible 基本上就等于白玩儿了。

如果 shell 模块能够讲话，Playbook 对它来说，真有点儿“既生瑜，何生亮”的感觉。shell 模块执行起来很快也很方便，而写一个 Playbook 似乎就不是那么简单的事情。我们又该如何看待 Playbook 呢？

相对于 Shell 脚本，Playbook 的优势在于幂等性。如果 Shell 脚本自身的校验逻辑不够严谨，胡乱地重复执行有可能会出现意想不到的问题。Playbook 则先天就自备检查功能。而且，Playbook 文件的可读性强，描述直观，一目了然，严格的语法检查保证了代码的规范性。其实，做到同样的幂等性和代码可读性，对于一个优秀的 Shell 程序员来说并非难事。但是你要花费一定的精力，并且用心才可以。而 Playbook 是靠内置幂等和强制规范来实现的，它的保证性更强。Playbook 更适合固化型的任务，而非灵活快速的调整。

#### 1. 第一个模板

我们很难在这里详细地讲解 Playbook 的全部语法。本节通过一些示例文件，给大家展示一些最常用的语句。

```
[root@station103 ~]# cat test.yml
---
- hosts: ip
  tasks:
    - name: file sharing & tigervnc
      yum: name={{item}} state=present
      with_items:
        - vsftpd
        - lftp
        - tigervnc
    - name: service
      service: name=vsftpd state=started enabled=yes
```

Playbook 文件的第一行都是以 --- 开头的，很像 Shell 里面的 shebang 声明。第二行的 - hosts: 用于组织主机节点，指定哪些被管理主机才会去执行这个 Playbook。第三行的

tasks：代表一个任务，它的下面可以包含多个 Actions，每一个 Action 都要使用 - name 分隔开来。如果把代码中的第二个 - name 注释掉，执行时就会出现如下错误。

```
[root@station103 ~]# ansible-playbook -C test.yml
ERROR: multiple actions specified in task: 'service' and 'file sharing & tigervnc'
```

## 2. 代码缩进

Playbook 使用的是 YAML 格式。YAML 是依靠缩进来划分代码层次关系的，所以书写时对于缩进的要求非常严格。另外，缩进不支持 Tab，这一点很不爽。因为我无意之间总是会习惯性地敲入一两个 Tab。如果你在执行 Playbook 时发现了如下错误内容，请使用命令 cat -A 检查你的 Playbook 中是否包含了 Tab。

```
[root@station103 ~]# ansible-playbook -C test.yml
ERROR: Syntax Error while loading YAML script, test.yml
Note: The error may actually appear before this position: line 7, column 1
```

```
with_items:
    - vsftpd
^
```

如果执行 Playbook 时出现了上述错误，请检查代码中是否包含 ^I，^I 就是 Tab。

```
[root@station103 ~]# cat -A test.yml
---$
- hosts: ip$
  tasks:$
    - name: file sharing & tigervnc$
      yum: name={{item}} state=present$
      with_items:$
^I^I^I- vsftpd$
      - lftp$
      - tigervnc$
    - name: service$
      service: name=vsftpd state=started enabled=yes$
```

## 3. 模拟测试

选项 -C 用来测试模拟执行后的结果返回，但它不会真的执行。在写好一个 Playbook 时首先要进行模拟测试，待确认没有问题后才能正式发布任务。同样，这里也支持选项 --list-hosts，用于组织主机节点时对匹配结果做检查。

```
[root@station103 ~]# ansible-playbook -C test.yml --list-hosts
playbook: test.yml
play #1 (ip): host count=3
192.168.0.83
192.168.0.85
192.168.0.87
```

当然，测试模拟也有一定的局限性。例如，如下示例文件中，由于软件包没有安装，



服务管理这部分的测试是无法完成的。

```
[root@station103 ~]# ansible-playbook -C test.yml

PLAY [ip] *****

GATHERING FACTS *****
ok: [192.168.0.83]
ok: [192.168.0.87]
ok: [192.168.0.85]

TASK: [file sharing & tigervnc] *****
changed: [192.168.0.87] => (item=vsftpd,lftp,tigervnc)
changed: [192.168.0.85] => (item=vsftpd,lftp,tigervnc)
changed: [192.168.0.83] => (item=vsftpd,lftp,tigervnc)

TASK: [service] *****
failed: [192.168.0.87] => {"failed": true}
msg: no service or tool found for: vsftpd
failed: [192.168.0.85] => {"failed": true}
msg: no service or tool found for: vsftpd
failed: [192.168.0.83] => {"failed": true}
msg: no service or tool found for: vsftpd

FATAL: all hosts have already failed -- aborting

PLAY RECAP *****
to retry, use: --limit @/root/test.retry

192.168.0.83      : ok=2    changed=1    unreachable=0    failed=1
192.168.0.85      : ok=2    changed=1    unreachable=0    failed=1
192.168.0.87      : ok=2    changed=1    unreachable=0    failed=1
```

#### 4. Register

网络上很多文章都将 Register 翻译成“注册变量”，我想叫寄存器应该是更适宜一些的。它可以将执行命令的结果暂存到一个变量里。这很像计算器中的 M+ 不是吗？

由于有些任务需要特定的条件来触发，在执行之前需要先检查条件是否具备。我们可以将检查结果先保存到 Register，然后配合条件语句 when 通过搜索 Register 中的关键字来决定下一步的动作。

搜索查询分为精准查询和模糊查询。下面是两个不同的 Register 的示例文件。

```
// 查询 tmp 目录下是否只有名为 yum.log 的对象，为真则在 root 家目录下 touch 文件 1
[root@station103 ~]# cat register1.yml
---
- hosts: ip
  gather_facts: no
  tasks:
    - name: command
```

```

    command: /bin/ls /tmp
    register: result
- name: file
  file: path=/root/1 state=touch
  when: result.stdout == "yum.log"
// 查询 tmp 目录下是否包含名为 yum.log 的对象, 为真则在 root 家目录下 touch 文件 1
[root@station103 ~]# cat register2.yml
---
- hosts: ip
  gather_facts: no
  tasks:
  - name: command
    command: /bin/ls /tmp
    register: result
  - name: file
    file: path=/root/1 state=touch
    when: 'yum.log' in result.stdout'
```

我们看到它俩的唯一区别就在于：前者要求 stdout 中有且只有 yum.log 条件才能成立，而后者只要搜索到 yum.log 就可以了。stdout 可以使用 split 方法像 awk 那样去分割取值，但是在使用精确查询时一定要注意，输出结果必须是固定不变的。如果 file 是在 /tmp/ 目录下操作的，即便 Action 执行成功，也仅此一次。因为执行成功后，/tmp/ 目录下会新增一个名为 1 的新文件，再次执行时条件就不再匹配了。

## 5. 调试

选项 -v 可以打印出详细信息，如果你觉得还是不够，可以使用 -vv 或者 -vvv 输出更多的信息。不过调试时，组织节点要限制在一定范围内，和调试无关的节点不要加入进来，过度的输出无益于你的调试工作。

下面展示的这个示例，就是在分析为什么精确查询时 192.168.0.87 这台主机不会 touch 文件。

```
[root@station103 ~]# ansible-playbook -v register1.yml
```

```
PLAY [ip] *****

TASK: [command] *****
changed: [192.168.0.85] => {"changed": true, "cmd": ["/bin/ls", "/tmp"], "delta":
  "0:00:00.001544", "end": "2017-07-08 09:46:52.930507", "rc": 0, "start": "2017-
  07-08 09:46:52.928963", "stderr": "", "stdout": "yum.log", "warnings": []}
changed: [192.168.0.83] => {"changed": true, "cmd": ["/bin/ls", "/tmp"], "delta":
  "0:00:00.001618", "end": "2017-07-08 09:46:54.023835", "rc": 0, "start": "2017-
  07-08 09:46:54.022217", "stderr": "", "stdout": "yum.log", "warnings": []}
changed: [192.168.0.87] => {"changed": true, "cmd": ["/bin/ls", "/tmp"], "delta":
  "0:00:00.001670", "end": "2017-07-08 09:46:53.987050", "rc": 0, "start": "2017-
  07-08 09:46:53.985380", "stderr": "", "stdout": "yum.log\nyum_save_tx-2017-06-29-
  09-15hmTKsd.yumtx\nyum_save_tx-2017-07-07-13-21019ma5.yumtx", "warnings": []}
```

```

TASK: [file] *****
skipping: [192.168.0.87]
changed: [192.168.0.83] => {"changed": true, "dest": "/root/1", "gid": 0, "group":
    "root", "mode": "0644", "owner": "root", "size": 129501, "state": "file", "uid": 0}
changed: [192.168.0.85] => {"changed": true, "dest": "/root/1", "gid": 0, "group":
    "root", "mode": "0644", "owner": "root", "size": 0, "state": "file", "uid": 0}

PLAY RECAP *****
192.168.0.83           : ok=2    changed=2    unreachable=0    failed=0
192.168.0.85           : ok=2    changed=2    unreachable=0    failed=0
192.168.0.87           : ok=1    changed=1    unreachable=0    failed=0

```

## 6. 进退自由

有些 Action 执行会失败，导致后面的所有操作全部中断。例如，将上面的例子中的查询 /tmp/ 目录下的文件替换成查询一个不存在的文件。你就会看到如下错误。

```

[root@station103 ~]# ansible-playbook error.yml

PLAY [ip] *****

TASK: [command] *****
failed: [192.168.0.83] => {"changed": true, "cmd": ["/bin/ls", "/tmp/2"],
    "delta": "0:00:00.001495", "end": "2017-07-08 11:40:22.764367", "rc": 2,
    "start": "2017-07-08 11:40:22.762872", "warnings": []}
stderr: /bin/ls: cannot access /tmp/2: No such file or directory
failed: [192.168.0.87] => {"changed": true, "cmd": ["/bin/ls", "/tmp/2"],
    "delta": "0:00:00.001446", "end": "2017-07-08 11:40:22.790261", "rc": 2,
    "start": "2017-07-08 11:40:22.788815", "warnings": []}
stderr: /bin/ls: cannot access /tmp/2: No such file or directory
failed: [192.168.0.85] => {"changed": true, "cmd": ["/bin/ls", "/tmp/2"],
    "delta": "0:00:00.001398", "end": "2017-07-08 11:40:21.943937", "rc": 2,
    "start": "2017-07-08 11:40:21.942539", "warnings": []}
stderr: /bin/ls: cannot access /tmp/2: No such file or directory

FATAL: all hosts have already failed -- aborting

PLAY RECAP *****
                to retry, use: --limit @/root/tag.retry
192.168.0.83           : ok=0    changed=0    unreachable=0    failed=1
192.168.0.85           : ok=0    changed=0    unreachable=0    failed=1
192.168.0.87           : ok=0    changed=0    unreachable=0    failed=1

```

如果我们不想因为某一个 Action 的失败影响到其他 Action 执行，可以使用 `ignore_errors` 语句来处理。同样，你也可以使用 `failed_when` 语句来自定义退出条件。

```

- name: command
  command: /bin/ls /tmp/yum.log
  ignore_errors: yes

- name: command

```



```
command: /bin/ls /tmp
register: result
failed_when: '"yum.log" in result.stdout'
```

## 7. Tag

有时候我们只想让节点执行 Playbook 中的部分任务，总不能再重新编辑一个新的文件吧？使用 Tag 可以将 Playbook 切分成不同的部分。比较好的做法是为每一个 Action 或者一组类似的 Action 指定一个 Tag，就像下面这样。

```
[root@station103 ~]# cat tag.yml
---
- hosts: ip
  gather_facts: no
  tasks:
    - name: command
      command: /bin/ls /tmp
      tags: command

    - name: file
      file: path=/root/1 state=touch
      tags: file
```

在执行过程中使用选项 `--tag` 指定要执行的部分，使用选项 `--skip` 指定要跳过的部分，如果有多个 Tag 要被选中，请使用逗号进行分隔。实际效果如下所示。

```
// 执行 --tag="file", 则只会 touch 文件，而没有执行查询 tmp 目录的操作
[root@station103 ~]# ansible-playbook tag.yml --tags="file"
```

```
PLAY [ip] *****
```

```
TASK: [file] *****
changed: [192.168.0.85]
changed: [192.168.0.83]
changed: [192.168.0.87]
```

```
PLAY RECAP *****
```

```
192.168.0.83      : ok=1    changed=1    unreachable=0    failed=0
192.168.0.85      : ok=1    changed=1    unreachable=0    failed=0
192.168.0.87      : ok=1    changed=1    unreachable=0    failed=0
```

```
// 而执行 --skip="command,file", 则两个命令都没有执行
```

```
[root@station103 ~]# ansible-playbook tag.yml --skip="command,file"
```

```
PLAY RECAP *****
```

## 8. Delegate

Playbook 默认都是在本地执行，有时需要将命令重定向到另外一台主机上去执行。例如，在为一批新上架的服务器完成初始化部署后，将它们的名称解析添加到 DNS Server 里。

```
- name: Add RRs to DNS Server
  shell: 'echo "$HOSTNAME IN A $IPADDR" >> /var/named/d.zone'
  delegate_to: 192.168.0.73
```

## 9. 动态定义 Inventory

人的需求不同，就会站在不同的角度去看问题。如果你只负责某一个业务，那你就只会关心自己的那一亩三分地；如果你要负责整个底层架构，要考虑的问题就会很多。

我们前面说过编队的事情，SE 每天都面临着不同的编队组合，即使有正则表达式，执行任务依旧非常困难。比如，SE 收到了一封来自安全组的关于主机安全漏洞修补的邮件，需要处理的主机列表如下。

```
10.2.7.108
10.2.8.110
172.16.153.16
172.30.100.3
192.168.1.7
```

如果这样完全没有规律的主机有几十台怎么办？你可能有很多个数据中心，它们的地址段完全不同，或许它们都是一个业务，又或许它们之间根本没有任何关联。这种情况你怎么去做？

此时动态定义一个 Inventory 文件就很重要。我们在第 6 章里面讲过，服务器刚刚部署完毕时还没有将业务统计到里面来，这个状态下的服务器节点只作为一种资源池而存在。然而它作为被管理主机，已经被纳入 `/etc/ansible/hosts` 当中了。随着业务节点被应用部门申请后，我们在 CMDB 中将更新这台主机的业务字段。在执行配置管理操作的时候，允许通过接口调用从 CMDB 中查询该项业务所涉及的所有主机，将全部地址列表返回生成一个新的 Inventory 文件。Ansible 使用选项 `-i` 来指定一个特定的 Inventory 文件。

### 10.3.6 关于优化

由于 Ansible 是基于 SSH 协议的，在执行大批量任务时，时间成本是一个引人关注的问题。最大力度地优化性能是 Ansible 的核心任务，如果不能很好地解决性能问题，那么在大规模生产环境中就没有出场的机会了。

增加并发数量可以在一定程度上缓解性能问题。默认并发数是 5，这个数值过于保守，你可以适当提升并发的数量，但一定不要超过主机 CPU 的逻辑核心数。Ansible 主机应当只作为独立的服务器存在。以 Intel® Xeon® E5 2650 v2 为例，其逻辑核心数为 16，两路 CPU 加起来，Ansible 的并发数应当控制在 32 以下。没有必要增加更多的并发数，那样只会是徒劳的，因为过多地超过处理器核心数的 task，会引发系统之间大量的上下文切换，把资源白白浪费在内核态而非用户态之上。

前面我们已经谈到 SSH 的性能问题。强烈建议大家将 Ansible 管理主机的 OpenSSH 的

版本升级到 5.6 以上，并开启 `ControlPersist`。否则，每一次执行操作都需要重新认证，这是非常麻烦的，在大并发操作的情况下，会让执行时间大大延长。如果使用 `pipelining=True` 的方式则需要注释 `/etc/sudoers` 文件的 `Default requiretty`。优化工作不仅要开源，同时也要懂得节流。建议大家关闭 `gathering facts`。除非你真的需要去收集信息，否则没必要在每次执行操作前都做一遍无用功。

## 10.4 Puppet

说到 Puppet，我不得不提到它的老前辈 CFEngine。它的作者 Mark Burgess 希望用一个软件实现任务的自动化管理，其核心理念是让系统在任何情况下都能收敛到一个预定的理想状态。

Puppet 的作者 Luke Kanies 正是基于对 CFEngine 的研究开发了 Puppet。它是一种跨平台的配置管理工具，采用 C/S 的结构，让被管理主机周期性地向 Puppet Master 发送请求，以获得其最新的配置信息，并完成该配置信息的同步工作。这样的工作模式，不禁让曾经身为 Windows 管理员的我想到了 Windows Active Directory。

接下来，我就先为大家介绍如何快速地部署一个 Puppet 的基本模型，了解它是怎么运行的。然后再来讨论在实际环境中，有哪些亟待解决的问题，以及我们是如何解决的。

### 10.4.1 Puppet 快跑

Puppet 相对其他配置工具来说有点儿复杂。我们通过快速构建一个最简实例的方法，使 Puppet 先运行起来，让大家有一个感性的认识。

本次，我们只让 `redhat03.example.com` 一台主机作为被管理主机参与实验。构建一个最简 Puppet 模型一共需要五个步骤。

#### (1) 安装组件

使用 `yum` 在 `station103` 上安装 `puppet-server`，在 `redhat03` 上安装 Puppet。

#### (2) 配置 Agent

在 `redhat03` 上打开配置文件 `/etc/puppet/puppet.conf`，增加如下内容。

```
[main]
server = station103.example.com
[agent]
runinterval = 30
```

`puppet-server` 的安装基于 Puppet。有趣的是，两者共享同一个配置文件 `puppet.conf`。`[main]` 段是主要配置项，`server` 是用来指明 Puppet Master 是谁，Puppet 的配置通常采用 `hostname` 的风格来指明一台主机。因此，请注意一定要有 DNS 或者 `hosts` 确保主机名称的解析。`[agent]` 段是专门配置 Agent 参数的。Agent 默认每隔半小时会向 Puppet Master 同



步一次状态。这里我们把它修改成 30s，为的是能够尽快地看到结果，而不需要每次都在 redhat03 上手工强制执行一次命令。

### (3) 创建目录和模块文件

我们创建一个最简单的名为 test 的模块，目的是在 redhat03 的 /tmp/ 目录下面创建一个以主机名命名的 .tst 文件，其文件内容是 Testing！。

```
# touch /etc/puppet/manifests/{nodes.pp,site.pp}
# mkdir -p /etc/puppet/modules/test/manifests
# touch /etc/puppet/modules/test/manifests/init.pp

/etc/puppet/manifests/site.pp
import 'nodes.pp'

/etc/puppet/manifests/nodes.pp
node 'redhat03.example' {
    include test
}

/etc/puppet/modules/test/manifests/init.pp
class test {
    file { ["/tmp/$hostname.tst": content => "Testing!\n"] }
}
```

### (4) 启动服务

```
// 管理端
# service puppetmaster start
# chkconfig puppetmaster on
// 客户端
# service puppet start
# chkconfig puppet on
```

### (5) 签发证书

Puppet 服务启动后会以主机名为 CN 项来创建一个证书请求，这时需要 Puppet Master 为它签发这张证书，双方才能建立互信关系。使用如下命令完成证书签署。

```
# puppet cert sign redhat03.example.com
```

cert 命令用于证书管理。其中，list 是查看当前 Puppet Master 中的证书状态，sign 是签发证书，clean 是吊销证书。操作对象可以是证书请求，或者使用 --all 参数来代表所有条目。

在签署之前请使用 list 来查看当前的证书状态。如下面所示，如果一个条目的前面没有任何标记，说明该证书请求还没有被签发，双方的信任关系尚未建立；加号代表证书已签发，信任关系建立成功，Agent 可以接受来自 Master 方面的管控了；减号代表 Master 吊销了该 Agent 的证书，双方信任关系解除。

```
// 查看当前证书状态
```

```
[root@station103 ~]# puppet cert list --all
"redhat03.example.com" (68:17:F1:05:D3:92:CF:27:BB:71:82:C6:11:46:FC:B5)
+ "station101.example.com" (88:3F:BB:EB:CC:0F:3D:06:7A:D6:4E:39:93:8A:48:88)
+ "station103.example.com" (2D:53:84:D8:FD:CD:30:7B:64:70:B3:FD:D6:21:A1:2F)
// 签发一张证书请求
[root@station103 ~]# puppet cert sign --all
notice: Signed certificate request for redhat03.example.com
notice: Removing file Puppet::SSL::CertificateRequest redhat03.example.com at '/
var/lib/puppet/ssl/ca/requests/redhat03.example.com.pem'
// 第二次检查证书状态
[root@station103 ~]# puppet cert list --all
+ "redhat03.example.com" (15:C3:53:B0:50:27:7B:E3:47:A0:04:FA:63:FB:74:69)
+ "station101.example.com" (88:3F:BB:EB:CC:0F:3D:06:7A:D6:4E:39:93:8A:48:88)
+ "station103.example.com" (2D:53:84:D8:FD:CD:30:7B:64:70:B3:FD:D6:21:A1:2F)
// 吊销一张不再需要的证书
[root@station103 ~]# puppet cert clean station10[01].example.com
notice: Revoked certificate with serial 9
// 第三次检查证书状态
[root@station103 ~]# puppet cert list --all
+ "redhat03.example.com" (15:C3:53:B0:50:27:7B:E3:47:A0:04:FA:63:FB:74:69)
+ "station103.example.com" (2D:53:84:D8:FD:CD:30:7B:64:70:B3:FD:D6:21:A1:2F)
- "station101.example.com" (88:3F:BB:EB:CC:0F:3D:06:7A:D6:4E:39:93:8A:48:88)
(certificate revoked)
```

证书签发成功之后，我们再静待 30s，就可以看到如下结果了。

```
[root@station103 ~]# ansible 192.168.0.87 -a 'cat /tmp/redhat03.tst'
192.168.0.87 | success | rc=0 >>
Testing!
```

当然，如果你连 30s 的耐心都没有，也可以选择 **在 redhat03 上执行如下命令，立即实施同步操作。**

```
# puppet agent -t
```

## 10.4.2 初探 Puppet

很高兴我们没费吹灰之力就把 Puppet 给建立好了。本节我们来了解一下，Puppet 各组件结构之间的关系，以及使用 Puppet 时的一些入门经验。

### 1. 调用关系

初始安装结束后，Puppet 有 **modules** 和 **manifests** 两个空目录。**modules** 顾名思义就是模块，可以把它理解成函数。Puppet 把一组命令封装成一个 **module**，通常一个 **module** 就是一个具体的执行任务。**manifests** 里面存放的是配置文件，配置文件的扩展名是 **.pp**。

**manifests** 分为两个层次。第一个层次是 Puppet 这层的 **manifests** 目录，用来组织和引用全局的 **modules**。**site.pp** 是最初的起始文件，它用于初始化并引用其他所有的资源。早期版本，如果没有 **site.pp**，**puppetmaster** 服务甚至拒绝启动（**笔者**的版本没有 **site.pp** 文件依旧

可以顺利启动 puppetmaster 和 puppet 服务)。在上面这个示例里面，我们在 site.pp 上只做了一件事，就是把 nodes.pp 导入进来。

```
import 'nodes.pp'
```

nodes.pp 使用精确字符串或者正则表达式来完成不同主机节点的匹配。撰写 nodes.pp 可以被理解成在 Windows Domain Controller 中的 GPO 架构设计。nodes.pp 通过 include 语句直接引用 modules 中名为 test 的模块。

```
node 'redhat03.example' {
    include test
}
```

manifests 的第二个层次是 modules 下面的 manifests 目录，用来组织 modules 内部的一系列资源。test 模块下面需要建立一个 manifest 子目录，当这个模块建立完成后，至少要创建一个名为 init.pp 的文件，并在其中对 test 进行定义。

```
class test {
    file { ["/tmp/$hostname.tst": content => "Testing!\n" ]
}
```

注意：class 后面的名字必须和 module 的名字一致。花括号里面可以为空，代表仅仅定义了一个 module，但什么都没做。我们的例子是让 redhat03 在 /tmp/ 目录下面创建一个名为 \$hostname.tst、内容为 Testing! 的文件，其中 \$hostname 是一个内置变量，作用是获取 redhat03 的主机名。

好，现在我们了解了一件事，原来它们之间调用关系是这个样子的：site.pp → nodes.pp → test。接下来，我们在这个基础之上，把目录架构组织得更加规范一些。我们创建如下这样一个目录组织。

```
[root@cobbler ~]# tree /etc/puppet/
/etc/puppet/
|-- auth.conf          (验证配置文件)
|-- fileservers.conf   (允许哪些主机访问模块、文件和插件等)
|-- manifests          (存放配置)
|   |-- modules.pp     (导入模块，非必需)
|   |-- nodes.pp       (组织节点匹配类或直接引用模块)
|   |-- roles.pp       (创建一个类并且引用模块)
|   |-- site.pp        (初始化并引用其他资源)
|-- modules            (存放模块)
|   |-- base
|   |   |-- file       (存放文件)
|   |   |   |-- import.conf
|   |   |   |-- manifests
|   |   |   |-- init.pp
|   |-- test
|       |-- files
```



```
| | | `-- hello.txt
| | | `-- manifests
| | | `-- init.pp
|-- puppet.conf (puppet 配置文件)
```

在第一层 manifests 目录中我们新增了两个 pp 文件，同时修改了原 pp 文件的内容。

site.pp 引用了更多的 pp，还定义了执行命令的环境变量。

```
import 'modules.pp'
import 'roles.pp'
import 'nodes.pp'
```

```
Exec { path => "/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin" }
```

nodes.pp 新增了一组主机，我们看到 redhat03.example.com 匹配的是 test，而其他 redhat\*.example.com 的主机匹配的是 produce。

```
node /redhat.*\.example\.com/ {
    include produce
}
```

```
node 'redhat03.example' {
    include test
}
```

在 roles.pp 中，produce 和 test 都被定义成了类，由它们负责引用 modules。

```
class produce {
    include base
}
class test {
    include test
}
```

在 modules.pp 里，使用 import 导入 modules 目录中的模块。

```
import 'test'
import 'base'
```

现在，我们发现新的调用关系变成了下面这个样子：site.pp → nodes.pp → roles.pp → modules.pp → modules/\*。

我们使用 site.pp 完成了一系列的初始化工作，导入 pp 文件并定义环境变量，这一点很像 /etc/profile。nodes.pp 用来组织主机组以匹配不同的策略，在增加或者调整策略的情况下，我们都需要修改这个文件，就好像在 Windows Domain Controller 里面调整 GPO 一样。roles.pp 用于定义一个类，并让 nodes.pp 来引用，同时它自己也要引用 modules.pp 中的内容，而 modules.pp 则负责导入 modules 目录中的真正模块。

另外，还需要说明两点。第一，roles.pp 在定义类的时候，首字母不能大写，但是引用

这个类的名字的时候，对大小写并不敏感。第二，`nodes.pp` 在匹配时，精确字符串的优先级高于正则表达式，同为正则表达式的话只匹配第一个符合的策略，后面其他的策略都会被 Puppet 忽略掉。

我们来看下面这个例子。为了方便观察，`produce` 和 `test` 两组策略（类）分别调用了模块 `base` 和 `test`，它们做的事情几乎一样，都是在 `/tmp/` 目录下创建 `$hostname.tst` 文件，只不过内容有所区别，分别是 `Base!` 和 `Testing!`。

```
// 当有精确匹配时优先级最高
node /redhat.*\.example\.com/ {
    include produce
}
```

```
node /redhat03\.example\.com/ {
    include test
}
```

执行结果如下。

```
[root@station103 ~]# ansible ip -a 'cat /tmp/*.tst'
192.168.0.83 | success | rc=0 >>
Base!

192.168.0.85 | success | rc=0 >>
Base!

192.168.0.87 | success | rc=0 >>
Testing!
```

当全部是模糊匹配时后面的会被忽略（尽管后面的表达式比前面的更贴近实际结果）。

```
node /red.*\.example\.com/ {
    include produce
}
```

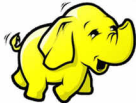
```
node / redhat.*\.example\.com/ {
    include test
}
```

执行结果是：

```
[root@station103 ~]# ansible ip -a 'cat /tmp/*.tst'
192.168.0.83 | success | rc=0 >>
Base!

192.168.0.85 | success | rc=0 >>
Base!

192.168.0.87 | success | rc=0 >>
Base!
```



## 2. 创建一个独立的 Test Zone

Puppet 的管理模式和 Ansible、SaltStack 有所不同，由于主机节点会主动上来同步配置，所以一个错误的配置会在同步周期之后给受影响主机带来难以估量的损害。

为了验证是否有效，可以在主机节点上使用 `--noop` 来测试。

```
# puppet agent -t --noop
```

但是，我并不推荐过度依赖 `--noop` 去验证。因为它只能证明你的 `pp` 文件的语法和执行是否能够成功，无法检查逻辑错误所带来的问题。而且，不管是否使用 `--noop`，只要 `runinterval` 的时间一到，被匹配的主机都会执行这个策略。

正确的做法是：在生产环境中构建一个独立的测试区用于模拟执行效果。为了节约资源，你可以使用虚拟机来充当测试节点，它们并不承担生产服务，只是作为模拟测试使用。

在完成一项策略配置后，将该模块首先挂载到测试区进行验证，待确认策略正确无误后，再将其挂载到需要执行该项操作的主机节点上去。

## 3. 逐步完善配置文件

事实上，Puppet 的配置文件的内容非常丰富，我们看到 `puppet.conf` 中的内容只是冰山一角而已。我们可以使用下面这些命令找到所有的配置项。根据你的实际情况，逐步完善并丰富自己的配置。

```
# puppetmasterd --genconfig
```

## 4. 资源管理

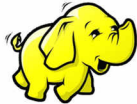
Puppet 的管理核心就是为由管理主机定义一个个配置项，这些配置项在 Puppet 中称为资源。你也可以把一个资源看成一个具体的策略。Puppet 的资源类型非常丰富。它的主要语法模式如下。

```
RESOURCE { ' OBJECT ' :  
    KEY => ' VALUE ',  
    KEY => ' VALUE ',  
    ...  
}
```

其中，`RESOURCE` 是一个资源类，主要可以分为系统（如 `file`、`exec`、`package`、`user` 等）和应用（如 `sshkey`、`nagios_host` 等）这两大类。`OBJECT` 是 `RESOURCE` 要操作的对象，如果 `RESOURCE` 是 `package`，那么 `OBJECT` 就是软件包的名字；如果 `RESOURCE` 是 `file`，那么 `OBJECT` 就是文件名。`KEY` 和 `VALUE` 也都是 `RESOURCE` 所拥有的属性和值。除了一些特殊的定义，大部分 `KEY` 对应的就是命令中的 `Option`（选项），`VALUE` 对应的就是 `Agruments`（参数）。

如果你想查询 Puppet 都含有哪些模块，可以使用下面这个命令。





```
# puppet doc |grep '^### '
```

下面是一些常用资源的示例。

#### 1) 软件部署:

```
package { 'tigervnc':  
  ensure => absent,  
  source => '/etc/yum.repos.d/cobbler-config1.repo',  
  provider => 'yum',  
}
```

#### 2) 用户管理:

```
user { 'luci':  
  ensure => present,  
  name => 'luci',  
  uid => '1999',  
  home => '/home/luci',  
  shell => '/sbin/nologin'  
}
```

#### 3) 文件管理:

```
file { ["/tmp/file":  
  owner => 'root',  
  group => 'se',  
  mode => '400',  
  content => 'File!'  
]}
```

#### 4) 命令执行:

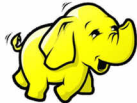
```
exec {'result':  
  cwd => '/tmp',  
  path => ['/usr/bin', '/bin'],  
  command=> 'cat /tmp/file > /tmp/result';  
}
```

### 10.4.3 使用 Apache + Passenger 替换 WEBrick

Puppet 默认使用内嵌的 WEBrick 提供给主机节点访问,但是 WEBrick 的并发性能无法用于生产环境。因此,将 Puppet 部署到生产环境后的首要任务就是用更加可靠的 HTTP Server 去替换 WEBrick,例如 Apache 和 Nginx。但是 Apache 和 Nginx 都无法直接为 Ruby 程序提供加载。这就需要使用 Passenger 将 Ruby 程序嵌入 Apache 或者 Nginx 的模块当中。实现 HTTP Server 的切换总共需要五个步骤。

#### 1. 安装 yum 源的软件包

```
# yum install -y httpd httpd-devel mod_ssl ruby-devel rubygems rubygem-passenger
```



## 2. 安装 gem 源的软件包

不使用 `-v` 会尝试安装最新的版本，如果相关组件版本不兼容会导致安装失败。此时可以尝试使用 `-v` 来指定软件的版本，具体版本需要根据实际情况而定。

```
# gem install rack -v 1.1.0
# gem install -y passenger -v 2.1.11
```

## 3. 配置 Apache

执行 `passenger-install-apache2-module`，第三步执行完毕后，程序会输出一份关于在 Apache 中如何载入 Passenger 模块的配置。我们将它复制到 `/etc/httpd/conf.d/10_passenger.conf` 文件里面。下面这个示例文件中的前三行就是 `passenger-install-apache2-module` 的输出结果，后五行是 Passenger 的其他配置项，你可以根据实际情况进行调整。

```
[root@station103 ~]# cat /etc/httpd/conf.d/10_passenger.conf
LoadModule passenger_module /usr/lib/ruby/gems/1.8/gems/passenger-2.2.11/ext/
  apache2/mod_passenger.so
PassengerRoot /usr/lib/ruby/gems/1.8/gems/passenger-2.2.11
PassengerRuby /usr/bin/ruby

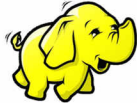
PassengerHighPerformance on
PassengerUseGlobalQueue on
PassengerMaxPoolSize 5
PassengerMaxRequests 3000
PassengerPoolIdleTime 1800
```

另外，我们还要创建 `/etc/httpd/conf.d/20_puppetmaster.conf` 文件，用于配置一个在 8140 端口监听的虚拟主机，用它来替换掉 WEBRick。

```
[root@station103 ~]# cat /etc/httpd/conf.d/20_puppetmaster.conf
<VirtualHost *:8140>
    ServerName station103.example.com
    SSLEngine on
    SSLProtocol ALL -SSLv2
    SSLCipherSuite ALL:!aNULL:!eNULL:!DES:!3DES:!IDEA:!SEED:!DSS:!PSK:!RC4:!MD5:+
      HIGH:+MEDIUM:!LOW:!SSLv2:!EXP
    SSLCertificateFile /var/lib/puppet/ssl/certs/station103.example.com.pem
    SSLCertificateKeyFile /var/lib/puppet/ssl/private_keys/station103.example.
      com.pem
    SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
    SSLCACertificateFile /var/lib/puppet/ssl/ca/ca.crt.pem
    SSLCARevocationFile /var/lib/puppet/ssl/crl.pem

    SSLVerifyClient optional
    SSLVerifyDepth 1
    SSLOptions +StdEnvVars +ExportCertData

    RequestHeader unset X-Forwarded-For
    RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
```



```

RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e

RackAutoDetect On
DocumentRoot /etc/puppet/rack/public/
<Directory /etc/puppet/rack/>
    Options None
    AllowOverride None
    Order allow,deny
    allow from all
</Directory>
</VirtualHost>

```

#### 4. 切换服务

配置完成后，停止 puppetmaster 并启动 httpd 即可。若要测试配置是否成功，你可以在主机节点上执行同步操作来完成验证。

```

// 管理主机切换服务
# server puppetmaster stop
# chkconfig puppetmaster off
# service httpd start
# chkconfig httpd on

// 主机节点验证
# puppet agent -t

```

### 10.4.4 Mutil-Master & Mutil-CAServer

以 Apache 或者 Nginx 的性能管理几千个节点是不成问题的。不过如果服务器数量进一步增加，单台 Puppet 不仅在性能上会产生瓶颈，而且在可靠性上也存在着很大风险。不管从哪个角度考虑，Puppet 集群模式都是实际生产环境中必不可少的。

创建多个 Puppet Master 需要使用负载均衡器实现前端的任务调度。这时，客户端是向负载均衡器发起 SSL 连接请求，负载均衡器要将 SSL 通信保持并传递到位于后端的 Puppet Master 上。否则，Puppet Master 就要卸载所有关于 SSL 的内容。

Puppet Master 默认是将 CA 和 HTTP 服务放置在一起的。如果使用多个 Puppet Master 会带来一个证书分散存储的问题。我们建议将 Puppet Master 中的 HTTP 和 CA 拆开，同样可以部署多个 CA Server，并使用共享存储集中存放所有 Puppet 的证书，实现 CA Server 的高可用。Agent 可以按照如下方式配置，实现 HTTP 和 CA 的分离。

```

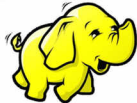
[agent]
server = puppetmaster.example.com
ca_server = puppetca.example.com

```

### 10.4.5 排障

Puppet 是一个非常稳定可靠的配置管理工具，不过由于它相对复杂的部署，总是容易





出现各种各样的问题。本节总结了一些常见故障的处理方法。

### 1. 无法解析主机名称

无法解析主机名称是部署 Puppet 时最有可能出现的一个错误。Puppet 系统中对于主机的名称解析是非常关键的一个环节，在实际生产环境中部署 Puppet 之前，首先要确保名称解析是有效的。不然，就会出现下面这个错误。

```
err: Could not retrieve catalog from remote server: getaddrinfo: Name or service
      not known
```

### 2. 信任关系未建立

信任关系未建立表示主机节点上没有找到签发的证书。证书是建立双方信任关系的凭证，需要在 Puppet Master 上签发证书请求。

```
Exiting; no certificate found and waitforcert is disabled
```

### 3. 证书验证失败

证书验证失败也是比较常见的。出现证书验证失败的原因有很多种。关于证书的具体细节，我们会在本书后面有关于安全的章节中进行详细说明。这里大家只要记住如下几个原因就行了。

- ☐ 证书有效期失效；
- ☐ 证书的 CN 项不匹配；
- ☐ 证书链不可信；
- ☐ 证书被吊销。

```
err: Could not retrieve catalog from remote server: certificate verify failed
```

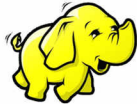
这个问题的根源分为两种情况：第一种是时间同步问题导致证书的签署时间和有效期不匹配所致；第二种是主机节点和管理主机上的证书不匹配所致，这种不匹配包括了证书的内容和证书的状态。

首先，我们要确认一下时间同步的问题。在阅读完第 9 章之后，我想处理时间同步对于我们来说应该不再是什么问题了。至于证书不匹配的错误，需要执行如下操作，通过重新签发证书来解除故障。

```
// 管理主机
# puppet cert clean < Error_Certificate >

// 主机节点
# rm -rf /var/lib/puppet/ssl/*
# puppet agent -t

// 管理主机
# puppet cert sign < New_Certificate >
```



#### 4. 类命名错误

类的首字母不能大写，Puppet 会提示这是一个语法错误。

```
err: Could not retrieve catalog from remote server: Error 400 on SERVER: Could
not parse for environment production: Syntax error at 'Produce' at /etc/
puppet/manifests/roles.pp:1 on node redhat03.example.com
```

#### 5. 对象未找到

其实类似于 Not found 这种故障是很容易解决的。如果你定义了一个模块或者类，至少应当创建一个对应的 pp 文件，并在其中定义一个空的类。另外，这里还要特别留意拼写问题，例如将 manifests 误写成了 mainfests。

```
err: Could not retrieve catalog from remote server: Error 400 on SERVER: Could
not find class test for redhat01.example.com at /etc/puppet/manifests/nodes.
pp:2 on node redhat01.example.com
```

#### 6. node 未匹配

node 未匹配是由于 note.pp 中定义的节点出现了匹配问题而引起的。node.pp 会按照你定义的字符串或者正则表达式去匹配 Puppet CA 中已经签发过证书的节点。如果没有找到，首先检查表达式是否正确，其次检查节点的证书是否已经签发。

```
err: Could not retrieve catalog from remote server: Error 400 on SERVER: Could
not find default node or by name with 'redhat01.example.com, redhat01.
example, redhat01' on node redhat01.example.com
```

#### 7. 证书含有非法字符

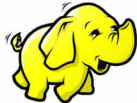
证书含有非法字符有可能在低版本的 Puppet Master 替换了 WEBRick 后于测试时出现。这是一个已知的 Bug，可以参考如下链接去处理：<https://projects.puppetlabs.com/issues/15561>。

```
err: Could not retrieve catalog from remote server: Certname "/c=CN/st=somestate/
l=somecity/o=someorganization/ou=someorganizationalunitstation103.example.
com/emailaddress=root@station103.example.com" must not contain unprintable or
non-ASCII characters
```

#### 8. 无法载入 passenger-install-apache2-module

无法载入 passenger-install-apache2-module 也是在 Puppet Master 替换 WEBRick 时出现的错误。这是由于在 gem 安装时没有指定正确的版本造成的，导致 passenger-install-apache2-module 没有正确链接应当执行的模块，要解决这个问题，可以在上一级目录下搜索正确的执行模块。

```
// 执行 passenger-install-apache2-module 后报错，返回没有对应的文件被载入。
[root@station103 ~]# passenger-install-apache2-module
/usr/bin/passenger-install-apache2-module:19:in `load': no such file to load --
/usr/lib/ruby/gems/1.8/gems/passenger-3.0.21/bin/passenger-install-apache2-
module (LoadError)
from /usr/bin/passenger-install-apache2-module:19
```



```
// 查询上一层目录中是否其他版本对应的模块
[root@station103 ~]# ls /usr/lib/ruby/gems/1.8/gems/passenger-2.2.11/bin/
passenger-install-apache2-module
/usr/lib/ruby/gems/1.8/gems/passenger-2.2.11/bin/passenger-install-apache2-module
```

## 10.5 SaltStack

我对 SaltStack 并不陌生，与它打交道的的时间不算短。它给我的印象很深，简单部署，轻松执行，而且非常快，很容易理解，几乎没有什么门槛。所以，我可能会有点儿偏心，把 SaltStack 的一些核心内容当作了本章的重点。好了，我们就一起进入 SaltStack 的世界吧。

### 10.5.1 配置 Minion

SaltStack 的角色划分非常形象。管理主机称为 Master，需要部署软件包 salt-master；被管理主机节点称为 Minion，需要部署软件包 salt-minion。事实上，SaltStack 在部署完毕后无须任何配置就可以开始工作。不过还是建议大家修改 Minion 配置文件 /etc/salt/minion 中的如下两行。

```
master: < SALT_MASTER_ADDRESS >
id: < SALT_MINION_IDENTITY >
```

master 用于指定 Master 的 IP 地址，如果这里不做配置，Minion 会尝试向一台名为 salt 的主机发送请求。id 是 Minion 的身份标识，默认使用 Minion 的主机名作为 id。不过这种形式的 id 不利于日后的操作，建议修改成 Minion 的 IP 地址。

### 10.5.2 管理 Salt Key

在 Minion 接受 Master 管理之前，必须经过 Master 的验证。salt-minion 服务在启动时，会生成密钥对和请求，Master 在通过请求后，会将自己的公钥发给 Minion。这个过程和 SSH 的公钥验证类似。

salt-key 命令用于管理 Master 下属所有 Minion 的添加和删除。

```
# salt-key -L 用于检查当前所有的 Salt Key
# salt-key -A 用于接受当前所有未验证的 Salt Key
# salt-key -D 用于删除当前所有的 Salt Key
# salt-key -a <ID> 指定接受一个未验证的 Salt Key
# salt-key -d <ID> 指定删除一个 Salt Key
# salt-key -l acc|grep -c -v 'Accepted Keys:' 检查现有 Minion 的数量
```

每当有新的 Minion 加入时，手工添加会显得比较麻烦，可以考虑在 Master 的配置文件中开启如下这行内容。



```
auto_accept: False
```

不过我的个人经验是最好手工并分批次执行 Salt Key 的验证。如果一次部署了大量的 Minion，自动添加所有 Salt key 可能会造成部分 Minion 的验证失败。如果 Minion 被拒或者长期无法和 Master 建立通信，到达超时设置后 salt-minion 服务会自动停止。

### 10.5.3 组织主机节点

相对于 Ansible，SaltStack 的表达方式要自然得多，有如下方式。

#### 1. 按组编队

按组编队需要事先在 Master 的配置文件 /etc/salt/master 中声明，文件修改后无须重启 salt-master 服务即可生效。

```
// 定义 192.168.0.83 和 192.168.0.85 为 Web 服务器，192.168.0.87 为 DB 服务器
nodegroups:
  web: 'L@192.168.0.83,192.168.0.85'
  db: 'L@192.168.0.87'
// 依照 web 分组和 db 分组，批量执行 ping 操作并返回结果
[root@station103 ~]# salt -N web test.ping
192.168.0.85:
  True
192.168.0.83:
  True
[root@station103 ~]# salt -N db test.ping
192.168.0.87:
  True
```

#### 2. 按照通配符或者正则表达式编队

SaltStack 默认使用的是 Shell glob 模式匹配，同时支持通配符和一些简单正则表达式的混用，不需要分得那么清楚。使用选项 -E 则会替换成 PCRE 模式，即 Perl Compatible Regular Expressions。

```
[root@station103 ~]# salt '*3' test.ping
192.168.0.83:
  True

[root@station103 ~]# salt '192.168.*.[3-5]' test.ping
192.168.0.83:
  True
192.168.0.85:
  True
```

### 10.5.4 模块的调用

和 Ansible 的 Ad-Hoc 类似，SaltStack 集成了很多模块供用户来选择。下面，我们就来介绍一些常用的模块。

### (1) 帮助

关于如何去使用模块，SaltStack 提供了清晰的帮助和示例。

```
# salt -d > salt.man
```

### (2) ping 模块

ping 模块是用来测试 Master 和 Minion 之间的连通性的。

```
# salt '*' test.ping
```

### (3) cmd 模块

cmd.run 是 SaltStack 的核心，它相当于 Ansible 中的 shell 模块。

```
# salt '*' cmd.run 'id'
```

如果要执行一个脚本。我常把它们部署到 HTTP 上面，然后这样来执行。

```
# salt '*' cmd.run 'curl -L http://station103.example.com/scripts/example.sh|bash'
```

不过有个缺陷是不便于携带参数。cmd.script 支持参数这一点很方便，而且它也支持 HTTP 协议。

```
# salt '*' cmd.script salt://scripts/example.sh "arg1 arg2 'arg 3'"
```

```
# salt '*' cmd.script http:// station103.example.com/scripts/example.sh "arguments"
```

### (4) pkg 模块

pkg 模块相比 yum 和 ansible 的回显结果，更加简单明了。

#### 1) 安装软件包 vsftpd 和 lftp。

```
[root@station103 ~]# salt '*' pkg.install 'vsftpd,lftp'
192.168.0.83:
```

```
-----
lftp:
```

```
-----
new:
```

```
4.0.9-1.el6
```

```
old:
```

```
vsftpd:
```

```
-----
new:
```

```
2.2.2-11.el6_4.1
```

```
old:
```

#### 2) 卸载软件包 vsftpd 和 lftp。

```
[root@station103 ~]# salt '*' pkg.remove 'vsftpd,lftp'
192.168.0.83:
```

```
-----
lftp:
```

```

new:
old:
    4.0.9-1.el6
vsftpd:
-----
new:
old:
    2.2.2-11.el6_4.1

```

### (5) file.replace

这个模块的文件替换效果很好，我们看到它很贴心地把修改结果都返回来了。就像在 ATM 机上取完现金后系统会告知你当前余额一样方便。有了这个模块，我就不太愿意用 sed 了。因为使用 sed 命令修改后，你不得不再次用 grep 命令去确认一下。

```

[root@station103 ~]# salt '192.168.0.83' file.replace /etc/ssh/sshd_config
pattern='#Port 22' repl='Port 22'
192.168.0.83:
---
+++
@@ -10,7 +10,7 @@
# possible, but leave them commented. Uncommented options change a
# default value.

-#Port 22
+Port 22
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

[root@station103 ~]# salt '192.168.0.83' file.replace /etc/ssh/sshd_config
pattern='Port 22' repl='#Port 22'
192.168.0.83:
---
+++
@@ -10,7 +10,7 @@
# possible, but leave them commented. Uncommented options change a
# default value.

-Port 22
+##Port 22
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

```

## 10.5.5 Mutil-Masters

只有一台 Master 的生产环境会给运维团队带来噩梦般的体验。前面我们提到过，如果 Minion 长时间无法和 Master 建立连接，那么它的服务将自动停止。想象一下如果单台



Master 突然挂掉了得有多么可怕。

如果你的管理环境中只有 SaltStack 一种配置管理通道，那么一定要配置 Mutil-Masters 来应对这个问题。创建 Mutil-Masters 并不困难，只要确保所有 Master 使用同一密钥即可。Master 存放密钥的目录位于 `/etc/salt/pki/master/` 下面，将 `master.pem` 和 `master.pub` 复制到每一台 Master 的同一目录下。Minion 则在配置文件 `/etc/salt/minion` 中如下指定 Mutil-Masters 的 IP 地址。

```
master:
- < SALT_MASTER_ADDRESS >
- < SALT_MASTER_ADDRESS >
```

这样配置完成后，Minion 会按照书写顺序先向第一台 Master 发送请求，待通过验证后，Minion 才会向第二台 Master 发送请求。

另外，需要注意一点的是：使用 `cmd.scripts` 执行脚本时，脚本一定要存放在 HTTP Server 上，而不是默认的 Master 本地的 `/srv/salt/` 目录下。使用 `salt://` 发布的话，Minion 会随机到一台 Master 上去取脚本，如果脚本只在一台 Master 上存放，那么随机到其他 Master 的 Minion 会因为找不到脚本而无法执行。没有必要特意去同步本地的 `/srv/salt/` 目录，使用 HTTP 协议才是更恰当的方案。

## 10.5.6 级联

有些 PE 或者 DBA 不愿意使用全局的 Master。他们给出的理由是不希望误操作，但我觉得根本原因不在于此，也许是编辑 NodeGroups 的不便或者其他深层次的理由。如果搭建私有化的 Master 又会引发管理孤岛；而且 SE 无法执行全量的操作。这时不妨考虑一下级联的模式。

如图 10-1 所示，Master 和 Minion 之间不再直接建立管理关系，两者之间增加了 Syndic 的角色。

启用级联模式时，Syndic 上面要安装软件包 `salt-master` 和 `salt-syndic`。然后在 Syndic 上修改配置文件 `/etc/salt/master`，指定自己的上级 Master。

```
syndic_master: < SALT_MASTER_ADDRESS >
syndic_master_port: < SALT_MASTER_
RET_PORT >
```

最后在 Master 上修改配置文件 `/etc/salt/master` 中的如下内容，让它知道自己下面管理的是 Syndic。

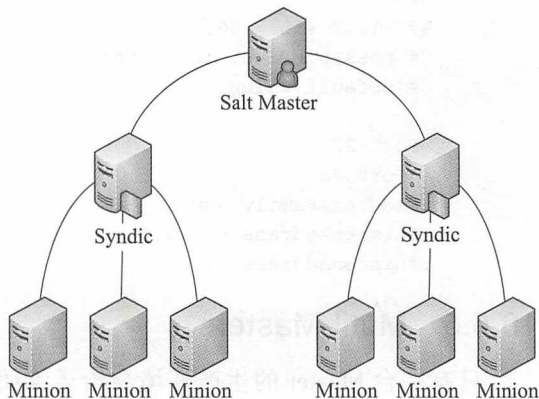


图 10-1 Syndic 架构

```
order_masters: True
ret_port: < SALT_MASTER_RET_PORT >
```

### 注意:

默认 `syndic_master_port` 的值和 `ret_port` 都是 4506。对于 Master 来说, Syndic 本质上也是 Minion, 只不过是高级一点儿的 Minion 罢了。这样一来, Syndic 的 4506 既要接收下面 Minion 的返回, 同时又要向上汇报给上级的 Master, 负担会比较大。建议将其更换到另外一个端口中去。但是请一定要确保 Syndic 的 `syndic_master_port` 和 Master 的 `ret_port` 是一致的。

还有, 如果使用级联模式会增加故障点, 推送任务出现问题后你不得不同时检查 Syndic 和 Minion。所以, 最好是可以将 Syndic 的服务权限委派给其他人员维护, 允许他们建立并维护自己的群组, 由 SE 负责维护顶层的 Master。

## 10.5.7 SLS

SLS 是 Salt State 的缩写。正如其名, 它是一个状态文件, 为的是让 Minion 达到一个状态, 并不关心中间是怎么操作的。SLS 使用的也是 YAML 格式, 但它理解起来要比 Playbook 容易得多, 而且 man 手册中有详细的教程。官方网址 <https://github.com/saltstack/salt-states> 中还提供了很多示例文件模板。

首先, 在 `/srv/salt/` 目录下创建一个名为 `top.sls` 的文件。`top.sls` 是最初的起始文件, 它用于初始化并引用其他所有的资源。文件内容类似于如下这样。

```
base:
  '192.168.0.8[35]':
    - match: pcre
    - vnc
    - lftp
```

`base` 称为环境, 你可以把它当成一个名字。第二行是匹配字符串。第三行是匹配模式, 不同的模式对第二行给出的字符串的理解是不同的。这里的示例指定了使用 PCRE 模式, 同时也支持 Shell glob 和 Grain。第二行与第三行合起来用于组织主机节点, 这样执行 SLS 的时候不用再去考虑编队的问题了。后面两行分别引入了两个 SLS 文件, 它们的内容如下所示。

```
// 安装关键包 tigervnc
tigervnc:
  pkg:
    - installed
// 安装关键包 lftp
lftp:
  pkg:
    - installed
```

它们的作用就是安装两个软件包。直接执行 `salt '*' state.apply` 后，除 192.168.0.87 以外，另外两台主机都会安装软件包 `tigervnc` 和 `lftp`。

## 10.5.8 Grain

Grain 类似于 Facter，是 SaltStack 预先内置好的一些变量，在这些变量后面通过脚本调用收集并存储 Minion 主机的特定信息。使用 `grains.ls` 可以看到总计共有 50 多项内容。

```
[root@station103 salt]# salt '192.168.0.83' grains.ls|wc -l
53
```

`grains.items` 可以看到所有内容。而 `grains.item` 则是用来指定查询某一个信息。`salt` 命令有一个选项 `-G`，是用来匹配 Grain 的，也就是说除了前面提到的两种编队模式，SaltStack 还支持根据主机的信息来进行编队。

```
[root@station103 salt]# salt '192.168.0.83' grains.item hwaddr_interfaces:eth0
192.168.0.83:
-----
hwaddr_interfaces:eth0:
    52:54:00:13:04:52

[root@station103 salt]# salt -G 'hwaddr_interfaces:eth0:52:54:00:13:04:52' grains.
item ip4_interfaces:eth0
192.168.0.83:
-----
ip4_interfaces:eth0:
    - 192.168.0.83
```

而且更妙的是，Grain 是可以自定义的，这很像 Ansible 的动态 Inventory。既然 Node-Groups 编写起来很麻烦，何不通过 Grain 来完成业务编队呢？

还记得第 6 章和第 7 章内容么？我们把服务器内部分类定义为 Classes，根据不同的服务器类型，部署不同的应用场景，也就是让 Classes 对应 Cobbler 中的 Profile。在执行完成后，我们可以将这个 Classes 类以文本方式写入系统其中的一个 info 文件里面。然后我们用 Classes 作为一个自定义 Grain，通过 `-G` 匹配服务器类型去执行不同的操作。除了 Classes，我们还可以将 Owner、业务名称等信息都写入 info 中去，全作为自定义 Grain，以便于日后 SLS 的执行。对于大多数 SE 来说，写一个 Python 脚本要比写 Facter 更加容易一些。

官方手册上关于自定义 Grain 给出的是这样一个示例。

```
#!/usr/bin/env python
def yourfunction():
    # initialize a grains dictionary
    grains = {}
    # Some code for logic that sets grains like
    grains['yourcustomgrain']=True
    grains['anothergrain']='somevalue'
```



```
return grains
```

不过，这种直接赋值的自定义对我们来说是没有什么意义的。我们希望通过写一个命令去取相应的值。下面是一个名为 `server_type` 的自定义 Grain 的示例代码。

```
#!/usr/bin/python
import os
def server_type():
    grains={}
    str=os.popen("awk -F: '{print $NF}' /root/info").read()
    grains['server_type']=str.strip()
    return grains
```

我们在三台主机节点上都部署了一个 `info` 文件，里面的内容如下所示。

```
[root@station103 _grains]# salt '*' cmd.run 'cat /root/info'
192.168.0.83:
    server_type:web
192.168.0.85:
    server_type:web
192.168.0.87:
    server_type:db
```

在 Master 的 `/srv/salt/` 目录下创建一个名为 `_grains` 的子目录，把刚才那个 Python 脚本放到里面，并同步到三个主机节点上去。

```
[root@station103 _grains]# salt '*' saltutil.sync_grains
192.168.0.83:
    - grains.server_type
192.168.0.85:
    - grains.server_type
192.168.0.87:
    - grains.server_type
```

然后我们来检查一下这个自定义的 Grain 是否生效了。

```
[root@station103 _grains]# salt '*' grains.item server_type
192.168.0.83:
    -----
    server_type:
        web
192.168.0.85:
    -----
    server_type:
        web
192.168.0.87:
    -----
    server_type:
        db
```

再用 `-G` 测试一下匹配是否成功。

```
[root@station103 _grains]# salt -G 'server_type:WEB' test.ping
192.168.0.85:
  True
192.168.0.83:
  True
```

```
[root@station103 _grains]# salt -G 'server_type:db' test.ping
192.168.0.87:
  True
```

从上面的测试结果中可以发现，Grain 的值是不区分大小写的。接下来，我们再给出如何自定义一个多值 Grain。注意：自定义多值的时候，不能使用元组的构造方式。虽然元组可以被读取出来，但在使用 -G 进行匹配时，Master 会出现 Minion did not return. [No response] 的错误。有兴趣的读者可自行测试。这一点可以参考 /usr/lib/python2.6/site-packages/salt/grains/core.py 中的代码，描述多值应当使用列表的方式来构造。

```
#!/usr/bin/python
import os
def mult_value():
    grains={}
    mult_value=['server1','server2','server3']
    grains['alias']=mult_value
    grains['bool']='True'
    return grains
```

为了让读者看得清楚，我在这里故意动了一些手脚，把上面这个示例做成两个不同的版本。第一个版本中的 mult\_value 包含 server1、server2 和 server3，并将其命名为 mult\_value\_web。第二个版本的 mult\_value 则只包含 server1 和 server2，并将其命名为 mult\_value\_db。将它们分开对送到两组主机上去。

```
[root@station103 ~]# salt '*'[35]' saltutil.sync_grains
192.168.0.83:
  - grains.mult_value_web
192.168.0.85:
  - grains.mult_value_web

[root@station103 ~]# salt '*'7' saltutil.sync_grains
192.168.0.87:
  - grains.mult_value_db
```

我们先调用 grains.item 检查一下自定义 Grain 是否已经生效。可以看到多值是用“-”来进行分隔的，而单值则直接被列举出来。

三台主机分别被赋予了不同的自定义 Grain 值。

```
[root@station103 ~]# salt '*' grains.item alias bool
192.168.0.87:
  -----
```

```
alias:
  - server1
  - server2
bool:
  True
```

```
192.168.0.85:
```

```
alias:
  - server1
  - server2
  - server3
```

```
bool:
  True
```

```
192.168.0.83:
```

```
alias:
  - server1
  - server2
  - server3
```

```
bool:
  True
```

然后再使用 -G 来测试匹配是否成功。

192.168.0.87 并不包含 server3，所以执行没有匹配到它。

```
[root@station103 ~]# salt -G 'alias:server3' grains.item server_type
```

```
192.168.0.85:
```

```
server_type:
  web
```

```
192.168.0.83:
```

```
server_type:
  web
```

在 Minion 的配置文件中也可以定义 Grain。注意：如果这里定义的 Grain 和 `_grains` 目录中定义的 Grain 有冲突，则会以这个 Grain 设定的值为准。

在 `/etc/salt/minion` 中自定义 grains

```
# Custom static grains for this minion can be specified here and used in SLS
# files just like all other grains. This example sets 4 custom grains, with
# the 'roles' grain having two values that can be matched against.
```

```
grains:
```

```
bool:
  - static1
  - static2
  - static3
```

// 在 `/etc/salt/minion` 中自定义的 grains 的优先级要高于 `_grains` 中自定义的 grains

```
[root@station103 ~]# salt '*' grains.item bool
```

```
192.168.0.87:
```



```

-----
bool:
  True
192.168.0.85:
-----
bool:
  True
192.168.0.83:
-----
bool:
  - static1
  - static2
  - static3

```

### 10.5.9 Pillar

Pillar 和 Grain 这两个名字起的实在是令人不知所云，很多初学者会把它们搞混也是很正常的。它俩看上去都是用来定义一个变量的，但两者的用法却截然不同。

Grain 是在 Master 上面创建的，但它的取值来自于 Minion，而且 Grain 的代码是通过调用模块 `saltutil.sync_grains` 将其同步到 Minion 上面执行的。在执行结果没有返回之前，Grain 的值是未知的，它要根据 Minion 的实际环境来确定。例如，OS 版本、硬件配置以及我们上一个例子中指定的服务器类型等。

和 Grain 不同的是，Pillar 是在 Master 上定义和存储的。Pillar 是为了在撰写 SLS 文件时便于引用而创建的，它的变量和赋值是在执行之前就已经确定好了的，和 Minion 的状态无关。

你可以这样来理解：把 Pillar 看成是预定义变量，而 Grain 则是用于存储返回结果的赋值变量。下面，我们借用官方给出的一个示例来展示如何使用 Pillar。

首先我们创建目录 `/srv/pillar` 和子目录 `/srv/pillar/users`，然后分别创建文件 `/srv/pillar/top.sls` 和 `/srv/pillar/users/init.sls`。

```

[root@station103 ~]# cat /srv/pillar/top.sls
base:
  '*':
    - users

[root@station103 ~]# cat /srv/pillar/users/init.sls
users:
  pe: 1001
  dba: 1002
  dev: 1003

```

这样，Pillar 就已经定义好了。接下来，我们就可以在 SLS 文件里面引用这个 Pillar 了。举个例子来说，系统部署完成后，有一步很重要的操作就是创建用户，我们可以在 SLS 文件中引用 Pillar 来完成这个创建用户的任务。

```
// 创建用户的 SLS
[root@station103 ~]# cat /srv/salt/users/init.sls
{% for user, uid in pillar.get('users', {}).items() %}
{{user}}:
  user.present:
    - uid: {{uid}}
{% endfor %}
```

注意：上面这个 SLS 要创建在位于 /srv/salt/users/ 目录下面。虽然名字都一样，但存储位置不同作用也完全不同。/srv/pillar/ 目录下面的 init.sls 是定义 Pillar 的，而 /srv/salt/ 目录下面的 init.sls 是创建用户的，它只不过在设置 UID 的时候引用了事先定义好的 Pillar 而已。

### 10.5.10 排障

我和 SaltStack 打了很长时间的交道，在使用过程中也遇到了不少奇奇怪怪的问题。本节将和大家分享一些 SaltStack 的故障处理案例。

#### 1. ulimit 文件打开数的限制

当一个 Minion 连接到 Master 上时，Master 至少要使用一个文件描述符，而这个数量默认受到 ulimit 的配置限制。如果你看到如下错误，有两个方法可以解决，要么调整 /etc/security/limits.conf 中关于 nofile 的数量，要么编辑配置文件 /etc/salt/master，打开 max\_open\_files 选项，设置一个合理的数值，它的优先级高于 ulimit。

```
2015-07-28 10:41:07,158 [salt.utils.verify] [[CRITICAL]
The number of accepted minion keys(3891) should be lower than 1/4 of the max
open files soft setting(4096). Please consider raising this value.
```

#### 2. 验证失败

SaltStack 的验证和 SSH 公钥验证类似，出现这样的问题，是因为 Master 拒签了 Minion，或者 Master 缓存的状态和 Minion 现有的状态不一致。一般来说最快速的解决方式就是重新验证。首先在 Master 上彻底删除关于这个 Minion 的 key 并重启服务，然后删除 Minion 节点上位于 /etc/salt/pki/minion/ 目录下的所有密钥，重启 salt-minion 服务后重新验证。

```
2015-08-03 16:22:48,943 [salt.crypt] [[ERROR ]
The Salt Master has cached the public key for this node, this salt minion
will wait for 10 seconds before attempting to re-authenticate
```

#### 3. Minion 启动调试

SaltStack 这个软件特别实诚，如果 salt-minion 服务启动是正常的，基本上不会出现太多的问题。出现故障时，有时服务一开始能起来，当到达超时时间后仍无法连接 Master 时才会停止（就像第二个问题）。而有时根本就启动不了，这时可以执行命令 salt-minion debug -l 检查调试信息。这里的 Debug 消息已经告诉我们处理这个故障需要重新验证。

```
[root@station103 salt]# /etc/init.d/salt-minion restart
Stopping salt-minion daemon: [FAILED]
Starting salt-minion daemon: [ OK ]
[root@station103 salt]# /etc/init.d/salt-minion restart
Stopping salt-minion daemon: [FAILED]
Starting salt-minion daemon: [ OK ]

[root@station103 salt]# salt-minion debug
[CRITICAL] The Salt Master has rejected this minion's public key!
To repair this issue, delete the public key for this minion on the Salt Master
and restart this minion.
Or restart the Salt Master in open mode to clean out the keys. The Salt Minion
will now exit.
```

下面是另外一个比较奇特的错误。这个问题和 SLS、Grain 等有关，简单地重新验证是没有效果的。要检查 Minion 中那些缓存的 SLS 或者 Grain，需要将它们一并删除。

```
[root@redhat01 ~]# salt-minion debug
[ERROR ] can't serialize <function st at 0x3479b18>
[WARNING ] ** Restarting minion **
[ERROR ] can't serialize <function st at 0x3479b18>
[WARNING ] ** Restarting minion **
[ERROR ] can't serialize <function st at 0x347ab18>
[WARNING ] ** Restarting minion **
...
```

#### 4. DNS 解析影响

salt 命令执行之前，都会先取得主机信息并尝试名称解析，如果 Master 主机在 /etc/resolv.conf 中指向了一个 DNS 服务，但是 DNS 不可用或者查询条目不存在，salt 会一直等到超时返回后再继续运行。如果你发现执行 salt 很慢，可以尝试先去执行 salt --version。这个命令和 Minion 没有关系，如果它执行返回都很慢，十有八九就是 DNS 解析惹的祸。

#### 5. 性能问题

很多性能问题的发现通常都是滞后的，节点数量少的时候大家都觉得还不错，等到规模增长之后，问题就会凸显或者放大。如果你的系统经常出现类似下面的这些错误，就说明性能瓶颈已经到来了。

示例 1:

```
2015-08-04 08:28:31,168 [salt.minion ] [WARNING ]
The minion failed to return the job information for job 20150804082529626151.
This is often due to the master being shut down or overloaded. If the master
is running consider increasing the worker_threads value.
```

示例 2:

```
Salt request timed out. The master is not responding. If this error persists
after verifying the master is up, worker_threads may need to be increased.
```



这个时期，大家会发现 SaltStack 时不常就会出现与 Minion 中断联系的情况，可是过一会儿它自己又好了。可以通过如下命令检查 Minion 的健康状态。

```
# salt-run manage.down
```

我曾经使用过 0.16、0.17 老版本的 SaltStack，其单台 Master 管理了超过了 1370 台的 Minion 依旧很稳定。后来从 2014 版本开始，尝试过多个版本，发现当 Minion 数量超过 2000 或者 3000 之后就会出现上述莫名其妙的问题。由于这个问题，我们有几次的任务执行都出现了严重的延迟现象，加上缓存中累积了太多的 Job 没有处理完，甚至有一次 Master 自己扛不住了，salt-master 服务直接就挂掉了。为了恢复 Master，我不得不删除掉队列中所有待执行的 Job（Master 的缓存目录位于 /var/cache/salt/master/ 下面）才清除了这个故障。

简单地提升 worker\_threads 是没有太多帮助的。一开始，我怀疑是版本升级的缘故。图 10-2~图 10-5 是 0.17.1 和 2015.5.2 进程树的差别。老版本上的子进程都是从一个父进程上 fork 出来的，从 2014 版本开始，父进程上 fork 出来的子进程再次 fork 出二级的子进程。

```
[root@station2000 ~]# salt --version
salt 0.17.1
[root@station2000 ~]# ps -C salt-master -o pid,ppid,user,cmd
  PID  PPID  USER      CMD
44236    1  root    /usr/bin/python /usr/bin/salt-master -d
44247 44236  root    /usr/bin/python /usr/bin/salt-master -d
44250 44236  root    /usr/bin/python /usr/bin/salt-master -d
44253 44236  root    /usr/bin/python /usr/bin/salt-master -d
44256 44236  root    /usr/bin/python /usr/bin/salt-master -d
[root@station2000 ~]# grep worker_threads /etc/salt/master
worker_threads: 1
[root@station2000 ~]#
```

图 10-2 SaltStack 0.17.1 单进程

后来，随着不断地学习研究，我意识到 3000+ 的节点数量对于基于 ZeroMQ 的 SaltStack 来讲压力太大了。

ZeroMQ 是一个简单快速的消息队列，它既不像基于 HTTP 架构那样，Minion 要定期去 Master 上去拉取任务，也不像 SSH 架构那般，让 Master 给每一个 Minion 去推送任务。ZeroMQ 采用广播形式将任务发布出去，Minion 通过 publish\_port（TCP 4505）接收 Master 布置的任务，并将执行结果通过 ret\_port（TCP 4506）返回。因为 ZeroMQ 总线是广播形式的，所有的 Minion 都共享一个消息通道，当消息过多时就会发生阻塞。

```

[root@station103 ~]# salt --version
salt 2015.5.2 (Lithium)
[root@station103 ~]# ps -C salt-master -o pid,ppid,user,cmd
  PID  PPID  USER      CMD
 43783      1  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 43784  43783  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 43785  43783  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 43786  43783  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 43789  43783  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 43792  43789  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 43793  43789  root      /usr/bin/python2.6 /usr/bin/salt-master -d
[root@station103 ~]# grep worker_threads /etc/salt/master
worker_threads: 1
[root@station103 ~]#

```

图 10-3 SaltStack 2015.5.2 单进程

```

[root@station2000 ~]# salt --version
salt 0.17.1
[root@station2000 ~]# ps -C salt-master -o pid,ppid,user,cmd
  PID  PPID  USER      CMD
 44650      1  root      /usr/bin/python /usr/bin/salt-master -d
 44651  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44658  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44659  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44674  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44675  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44678  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44681  44650  root      /usr/bin/python /usr/bin/salt-master -d
 44684  44650  root      /usr/bin/python /usr/bin/salt-master -d
[root@station2000 ~]# grep worker_threads /etc/salt/master
#worker_threads: 5
[root@station2000 ~]#

```

图 10-4 SaltStack 0.17.1 多进程

```

[root@station103 ~]# salt --version
salt 2015.5.2 (Lithium)
[root@station103 ~]# ps -C salt-master -o pid,ppid,user,cmd
  PID  PPID  USER      CMD
 6822      1  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6823  6822  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6824  6822  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6825  6822  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6828  6822  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6831  6828  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6832  6828  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6835  6828  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6838  6828  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6841  6828  root      /usr/bin/python2.6 /usr/bin/salt-master -d
 6844  6828  root      /usr/bin/python2.6 /usr/bin/salt-master -d
[root@station103 ~]# grep worker_threads /etc/salt/master
worker_threads: 5
[root@station103 ~]#

```

图 10-5 SaltStack 2015.5.2 多进程



新的解决方案是使用 RAET，它是基于 UDP 协议的。UDP 协议不关心握手和可靠性，所以它的速度处理非常快。不必担心加密和可靠性的问题。RAET 组件自带可靠性保障，并使用 libsodium 处理加密事宜。与 ZeroMQ 的模式不同，RAET 模式下，每个 Minion 都有独立的消息通道，Master 使用 RAET 发送任务时，并不会立即发生什么，而是将其放置在合适的堆栈中等待，并发调度会在各个堆栈中不断轮询并妥善处理。这种设计为 RAET 带来了非阻塞的性能，非常适合大规模任务发布。

启用 RAET 的前提是确保 libsodium、raet、ioflo、libnacl 这几个包都正常安装。libnacl 负责处理加密，而 ioflo 负责队列和流控。libsodium 是 libnacl 调用的动态库。

配置 RAET 几乎不需要做什么变更，只需将下面这行配置写入 Master 和 Minion 的配置文件中并重启服务就可以了。

```
transport: raet
```

不过在实际实施过程中，要注意版本之间的兼容性。大部分管理员习惯通过 yum 完成软件安装，而有些软件则必须通过 pip、easy\_install 或者其他方式引入，基于生产的 yum 源往往滞后于很多基于开发的软件源，从而引起版本不兼容的问题。如果在重启 salt-master 或者 salt-minion 服务时出现问题，请注意检查上述软件是否安装，并确保版本的正确性。下面是我曾经遇到过的一些错误。

由于缺少 raet 软件包所引起的错误。

```
ImportError: No module named raet
```

由于缺少 libsodium 软件包所引起的错误。

```
OSError: Could not locate nacl lib, searched for libsodium.so, libsodium.so.18,
libsodium.so.17, libsodium.so.13, libsodium.so.10, libsodium.so.5, libsodium.so.4
```

由于找不到对应版本动态库中的符号定义所引起的错误。

```
AttributeError: /usr/lib64/libsodium.so.4: undefined symbol: crypto_verify_64_bytes
```

## 10.6 我们真的能抗住海量节点吗

其实，写到这里的时候，我的心情是复杂的。我先后经历过 Puppet 和 SaltStack，有一段时间因为 SaltStack 的问题，很多同事都去研究 Ansible。类似的情况在很多家公司都在不同程度地上演着。关于开源工具的使用，往往最后都形成了两个极端。要么抱着原生配置用，发现不行了就换另外一个。或者干脆自己重新造轮子。事实上，我觉得能作为开源产品发布并拥有比较大的用户群，都不会是一个特别差的产品。主要问题还是在于我们自身对一个产品的应用场景缺乏深入了解，出现问题时没有认真地研究问题，一味地推翻重来而非去改善现有方案。这些问题都值得我们去思考。下面我介绍一些我思考过的问题。



### 10.6.1 集合编队

前面我们在介绍 Ansible 的时候，提到过实现动态 Inventory。不单单是 Ansible，这个功能对于所有快速响应的配置管理工作来说都是非常重要的。在 SaltStack 的章节中我们介绍了如何结合 IaaS 系统来自定义 Grain，从而实现了类似的动态 Inventory 的模式。但不管怎样实现，都离不开一个重要的因素，那就是你的基础架构平台的构建。

**试想：**如果没有 CMDB，你的信息从哪里来？没有规范的 Workflow，信息如何确保准确？如果各个系统接口的消息返回不能实现统一，接入一个新的系统后，是不是又得重新来过？磨刀不误砍柴工，把精力多用在基础架构的建设上是不会吃亏的。

### 10.6.2 汇报战况

如果是几十个管理节点，那么执行结果基本上看返回或者日志就可以了。如果是成千上万的节点，那么必须要具备汇总的功能。否则我怎么知道哪些节点执行成功了，哪些节点有问题需要去 retry？Ansible、Puppet 和 SaltStack 都有返回结果的解决方案。

以 SaltStack 为例，可以使用 `--return` 参数来指定新的定向通道。比较常用选项的是 `syslog` 和 `mysql`。`syslog` 会将执行结果返回到本地的 `messages` 日志当中，而 `MySQL` 则会将结果录入数据库。`MySQL` 需要的前置条件在其代码注释中有详细说明，它们位于 `/usr/lib/python2.6/site-packages/salt/returners/` 目录下面，可根据实际情况进行调整。

假设在 Master 上执行如下命令：

```
# salt-call '*3' test.ping --return=syslog
```

则 Minion 上的 `/var/log/messages` 下面会记录执行结果：

```
Jun 21 09:40:06 redhat01 python2.6: salt-minion: {"fun_args": [], "jid":  
"20170621094007126642", "return": true, "retcode": 0, "success": true, "fun":  
"test.ping", "id": "192.168.0.83"}
```

其中，`return` 是执行输出，`fun` 是调用模块，`fun_args` 是模块的执行参数。因为示例中调用的是 `test.ping`，所以这里 `fun_args` 的取值为空。如果调用 `cmd.run`，那么 `fun_args` 的值就是需要执行的命令。

有了这些内容我们就可以实现任务结果的消息汇总。如果将消息存入数据库，就需要一个前端 Portal 来支持。如果以 `syslog` 的形式发送，可交给 `ELK Stack` 来处理。

### 10.6.3 不必过度依赖模块

虽然每个配置管理工具都自带了很多功能模块，但那些现成封装的模块未必都是好的。如果你想深入学习一款配置管理工具，不如好好研究一下 Ansible 的 Playbook、Puppet 的 DSL 或者 SaltStack 的 SLS。

编写模块的根本意图在于跨平台管理。例如，Debian 和 CentOS 的安装分别基于 APT 和 YUM，CentOS 7 和 CentOS 6 的服务管理模式也不同。如果不是出于这个目的，模块的优势就不是那么明显了。毕竟大多数命令在众多 Linux 甚至是 Unix 平台上面都是兼容的，而且有些模块不像命令那样支持多个参数，更何况模块要输入的内容一点儿也不比命令少，语法还不如命令记得牢。能用命令轻松解决的问题，何必多此一举去使用模块呢？茴香豆的“茴”字有四种写法，但是我们会写一种就可以了，没有必要浪费精力全都学会。

记得那是 2014 年，有一次我在数据中心干活的时候，偶遇同在那里的另一家互联网公司的 SE。我们相互之间随意攀谈了几句，无意间说到了配置管理。当他听说我们正在用  $\times\times$  时，忍不住吐槽说：“ $\times\times$  啊，我们也在用，但是它不好用啊。我用它下发文件时效率太低了，每当节点一多， $\times\times$  就响应不了了。”我说没有那回事情，他表现得很诧异。因为他们公司的服务器比我们还要少大概 1000 台。他一直摇头表示不信，说  $\times\times$  的文件分发模块肯定是有问题的。我一听他说这话，立刻就明白了。我问他为什么要用模块做文件分发呢？他说既然有这个模块我肯定要用的啊。听完这话后，我给了他一个建议——尝试使用 wget。

使用  $\times\times$  时，我从来就没考虑过那些花哨的模块，与其调用文件分发模块不如直接执行 wget 来得实在。不管怎样，使用内置模块都不可避免地要和管理端服务器发生交互。像文件分发这种任务，如果文件传输是基于模块的，那么开销肯定非常大。Python 也好，Ruby 也罢，请问哪一个会比 C 语言的执行效率更高？尤其是当你只有一个管理服务器的時候，这种效率问题会更加明显。

我从来不用担心 Apache 或者 Nginx 会出现严重的性能问题，因为它们的使命就是为了应对高并发访问的。我们的生产环境在批量部署 KVM 虚拟机的时候，底层全都采用 wget 去下载虚拟机模板。一个模板有几个 GB。在千兆网络内，服务器同时接受几十个进程的并发，下载速率依旧可以接受。不管怎么说，一个配置文件要比虚拟机模板小得多，直接批量执行 wget 不就好了吗？又何必费劲去调用模块呢？

这个例子证明了一个道理，类似于下载这种重型任务不要依赖所谓的专有模块。拆解任务是首要的，把开销大的操作重定向给系统命令去做，配置管理只承担轻量级的任务。这个原则对于海量节点的配置管理是非常重要的。同样它还适用于代码的编写。过于复杂的逻辑判断不要写在模块里面，完全可以把多余的东西放到一个脚本中，执行时先下载脚本，然后再在客户端本地执行脚本。切忌把工作都压在管理服务器这一台主机上面。

## 10.7 解决方案的选择

在选择解决方案之前，我们要从管理成本、可扩展性和执行效率这几个方面去评价一个配置管理工具的优劣。每个配置管理工具都有自己的特点和使用的应用场景。没有最好

的，只有更适合的。

如果一个工具过于复杂，不论是学习或者执行起来都比较吃力，那么高昂的管理成本将难以适合日常快速响应的工作。配置管理节点数量的规模通常都很大，可扩展性决定了这个工具最大的支撑能力。并发支撑应当永远跑在业务增长的前面，不能让扩展性成为性能瓶颈。如果并发支撑抗住了业务的增长量，但是执行效率不足依旧没有任何意义。因为每天都有大量的任务要去执行，执行效率会继可扩展性之后成为新的性能瓶颈。

Expect 和 Parallel SSH 都是最简单的执行工具，它们的优势在于完全没有学习成本，而且 Expect 几乎不会受到任何限制，只要 SSH 能登录就好。不过它只能执行一些简单的任务，执行效率也不高。在没有其他管理配置工具而不允许安装软件的情况下，如果你遇到了一些无聊琐碎的小任务，可以使用我前面为大家提供的模板脚本 `expect_model.sh` 来救急。

Parallel SSH 比起 Expect 来说，就是不用费力去修改 Expect 的脚本，而且是并发执行的，效率会高一些。不过它的并发经常导致输出乱序，而且比任何一个并发工具都严重。不光是节点之间顺序被打乱，连节点内的返回值都不能好好地 and 输出结果排列在一起。

```
[root@station101 ~]# pssh -P -A -t 1 -h iplist 'id'
Warning: do not enter your password if anyone else has superuser
privileges or access to your account.
Password:
192.168.0.85: uid=0(root) gid=0(root) groups=0(root) context=unconfined_
u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[1] 11:00:41 [SUCCESS] 192.168.0.85
192.168.0.87: uid=0(root) gid=0(root) groups=0(root) context=unconfined_
u:unconfined_r:unconfined_t:s0-s0:c0.c1023
192.168.0.83: uid=0(root) gid=0(root) groups=0(root) context=unconfined_
u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[2] 11:00:41 [SUCCESS] 192.168.0.87
[3] 11:00:41 [SUCCESS] 192.168.0.83
```

当然，这还不是最重要的。Parallel SSH 和 Expect 一样都是通过读取 Inventory 文件来实现批量管理的，无法像 Ansible 或者 SaltStack 那样使用丰富的表达式完成编队。

为此，我用 Python 写了一个样例脚本，通过调用 `re.findall()` 函数来实现这个功能。在 Python 提示行下，使用 `help(re.findall)` 可以查到 `findall` 方法的示例。

```
findall(pattern, string, flags=0)
```

下面这段代码基于 Python 2.6.6。

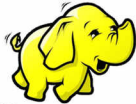
```
#!/usr/bin/python
```

```
# 导入正则表达式
```

```
import re
```

```
str=''
```





```
# 读取文件 iplist 中的所有地址
for line in open("iplist"):
    print line.replace('\n', '')
    str = ','.join([str,line.strip()])

print "-----"
print "Full IP List:", str.strip(',')

# 使用正则表达式, 列出符合 192.168.0.8[3-5] 的地址
list=re.findall("192.168.0.8[3-5]",str.strip(','))
print "Screening Result:", list
```

脚本的执行结果如下所示。

```
[root@station103 ~]# python regexp.py
192.168.0.83
192.168.0.85
192.168.0.87
-----
Full IP List: 192.168.0.83,192.168.0.85,192.168.0.87
Screening Result: ['192.168.0.83', '192.168.0.85']
```

在《运维前线》一书的发布会上,我听过松涛兄<sup>①</sup>分享的 Ansible 技术精髓。关于 Ansible 的优缺点,我个人认为其实都是源于 SSH 协议本身。它既是 Ansible 的优势,同时也给 Ansible 带来了一些难以回避的问题。

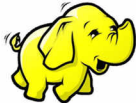
Ansible 采取了和 Parallel SSH 一样的策略,就是基于 SSH 协议通道来实现。SSH 协议的优势非常明显。首先,不需要操心终端那头,专心做自己就好。其次,相对于 Puppet 和 SaltStack 的 Agent,SSH 的稳定性好太多了。毕竟 C 语言的历史源远流长,在底层编译后,其软件的鲁棒性绝不是 Ruby 和 Python 所能比的。更何况 SSH 协议还自带加密光环,安全性也能保障。很多人认为 Ansible 的诞生和发展乃至成功,在很大程度上正是针对 SaltStack 的。但我觉得,Ansible 作为后来者,它并不只是简单地增强了 Parallel SSH 而已,它综合了 Puppet 和 SaltStack 的很多特性,既有类似于 Puppet 的功能强大的 Playbook,也有类似于 SaltStack 的 Ad-Hoc。能够博取众家之长,也许才是它能够逐步流行的主要原因。

同时,Ansible 的问题也来恰恰自于它的优势。首先,基于 SSH 服务始终会被各种连接和性能问题所困扰。其次,很多企业的 SSH 不允许 root 登录,只能使用 sudo,一些优化方案则是不支持 sudo 的。此外,因为密码过期或者各种人为因素导致 SSH 无法登录也是令人头疼的问题。

Ansible 引以为傲的 Playbook 虽然功能强大,但是其语法表达不如 SLS 更加平易近人。复杂场景下,对缩进的理解不到位就很容易迷茫,拿不准一条语句应当位于哪一个层次。另外,它的一些书写逻辑是倒序式的,比如 when、tag 等。还有就是一些语法的错误返回

---

① 李松涛,《Ansible 权威指南》一书第一作者。



提示得不够清晰。这对于初学者来说，上手还是很吃力的，使用 Ansible 的管理员更多是倾向于它的 Ad-Hoc。

Puppet 是一款久经考验的配置管理工具。Puppet 的用户群众多，在很多大型 IT 公司内有着成熟的管理模型和维护经验，可扩展性方面完全不必担心。它很靠谱儿，不会无缘无故地丢失任务，而且很像 Windows Active Directory 中的 Domain Controller。Puppet 有成熟的 Dashboard 方案，DSL 文件易于理解，而且网上有足够数量的开发模板。不过其缺点也很明显。

- ❑ 它不适合快速部署，你得慢慢等结果。至于 MCollective，它只能完成常见的任务，不够灵活，很多还需要自己去写。
- ❑ 凡事最终离不开 Ruby。Puppet 是以 Ruby 语法作为基础的，上手很容易，但是深入起来有难度。要是你对 Ruby 停留在一知半解的层面上，肯定是搞不好 Puppet 的。从零基础开始学习 Ruby 是件令人不开心的事情，至少我是这样认为的。如果你深入学习一门语言不是为了转型程序员，而只是为了系统管理，肯定是一件亏本的事情。

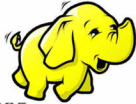
怎么去评价 SaltStack 呢？有人说，Puppet 是程序员的运维工具，而 SaltStack 才是真正运维人的运维工具。我想这样讲也是有些道理的。SaltStack 对比 Ansible 和 Puppet 来说有两个优势：第一，它不依赖于 SSH 服务，因此不受 sudo、账户过期、并发性能和 SSH 服务通道失效等一系列的约束。第二，它的表达式简单便捷、清晰明了，既能快速地执行命令，也可以非常轻松地实现状态管理，SLS 文件更加容易理解，其学习成本比起 Playbook 和 Ruby 来说要低得多。它的缺点在于并发瓶颈，在超过 3000 节点的江河模式场景下，ZeroMQ 的消息队列阻塞是无法令人满意的。你可以尝试最新版本并替换成 RAET。截稿之前，我已经看到 2017.7 版本的发布了。至少，SaltStack 作为小环境下的分治管理还是非常优秀的。

谈完了众多配置管理工具的优缺点，我们发现单一使用任何一个工具都是不理想的。从管理通道的冗余性以及应用场景的多变性上看，将它们组合起来应用会更加合理。下面，我为大家推荐一些组合形式。

#### （1）Parallel SSH + Expect

这两个工具都是无客户端，安装方便，几乎没有什么依赖关系，甚至 Expect 就是自带的。它们的优势在于高度的兼容性，你完全可以在自己的笔记本电脑上部署这样一台 Linux 虚拟机以备不时之需。这个方案对于做乙方的同学来说是最适合不过的了。因为客户的系统环境上是不能随便安装软件的，而且网络访问也会受限。只要 SSH 服务不死，它俩都可以工作得很好。平时可以使用 Parallel SSH 来工作，特殊情况可以让 Expect 来救驾。只要你使用它们的时候拥有着坚定不移的决心和吃苦耐劳的意志，除了操作效率低点儿以外，我竟然还真想不到它们有什么缺点。





### (2) SaltStack + Parallel SSH

SaltStack 很轻, Parallel SSH 又是无客户端的, 它俩在一起不会占用终端太多的资源。SaltStack 的执行效率高, 不用担心 SSH 的各种烦恼。如果 Minion 服务偶尔挂掉, 可以用 Parallel SSH 来“补刀”。遇上执行大规模的全量操作时, 也可以考虑让 Parallel SSH 上阵。

### (3) Ansible + SaltStack

这个组合是上一组的增强版本, 两个都是 Python 系的, 一个基于 SSH, 一个基于 Agent。即便你不是非常精通 Python, 同时使用它们的学习成本也很低。Ansible 的 Playbook 主打静态化的全量部署, SaltStack 则用于执行平日里需要快速响应的任务。

### (4) Puppet + SaltStack

和上一组的分工很类似, Puppet 负责 Ansible 的工作。两个工具都有 Agent, 不受 SSH 限制。Puppet DSL 要比 Ansible Playbook 更容易理解, 网络上可借鉴的经验很多, 处理问题也更高效。Puppet 的解决方案很成熟, 疑难杂症较少, 完全不用担心规模的问题。而且, Puppet 的维护模式是状态收敛, 可靠性很高。只要网络连通性没有问题, 在服务通信正常的情况下, 不用担心有漏网之鱼, 也不用担心配置被人篡改。不过, 被管理主机要同时安装两套 Agent, 既要安装 Python, 又要安装 Ruby, 承载较重会占取更多的系统资源。

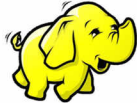
不推荐大家使用 Puppet + Ansible 的组合, 主要问题在于管理成本太高。两者之间有一些重合的地方, 不能实现取长补短的效果。Ansible 的精华在于 Playbook, 它的学习成本是不能避免的, 而且它又不能完全取代掉 Puppet。至于 Puppet DSL, 虽然它易于理解, 但是随着时间推移, 你会发现越来越多的需求变更触碰了你的知识盲区。不论是 Playbook 还是 Ruby, 为了处理这些问题, 你不得不花费大量的时间去进一步学习。若两个工具同时使用精力牵扯过大, 既不现实也没必要。如果放弃 Ansible Playbook, 那么还不如直接选择 SaltStack。

## 10.8 本章小结

本章旨在抽取各类配置管理工具中最为关键的部分, 为读者构建一个最小化的运行环境, 让大家有一个初步的感性认识, 了解它是怎么运行的。然后再来讨论它们是如何与实际环境相融合的, 还分享了一些解决方案和故障排错的示例。最后, 笔者根据自己的工作经验, 为大家总结了各类配置管理工具的优缺点和适用场景, 以及配置管理工具的选型方案。

下一章将是基础服务篇的最后一章, 届时将为读者介绍几种常见的文件共享服务的构建方案。





## 第 11 章

# 文件共享服务

本章作为基础服务篇的最后一章，将为读者介绍几种常见的文件共享服务的构建方案。生产系统少不了文件共享服务的支持。有些服务用于提供诸如软件、驱动、代码等资源的发布，有些服务则用于数据交换或分发。不同的应用场景其实现形式也各不相同。下面，我们就分门别类地一一道来。

### 11.1 构建 WebDAV 服务

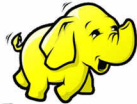
运维工作也算是一种服务行业。要想做好对外服务的工作，首先要把自己的内务打理好。我们在本书的第 3 章中介绍过，数据中心有很多区域，其中最重要的就是管理区。管理区是生产环境中的核心区域，所有的基础服务都部署在这里。管理区内有很多资源，例如软件仓库、管理工具、驱动程序、各种补丁以及常用的代码脚本等。它们通常都会被存放在 HTTP Server 上，用户可以通过 wget 下载相关的资源。除了下载以外，我们也有很多上传的需求。比方说，一台服务器出现了硬件故障，SE 要报修这台服务器，需要收集相关的信息，但安全策略不允许用户直接从服务器上下载数据，须先将数据上传到文件服务器，然后再统一下载。这时的文件共享服务就起到了中转站的作用。还有，我们日常开发的一些脚本或工具，希望提供给大家使用，也会用到上传的功能。

HTTP 协议本身并没有提供上传的接口，这就要借助 WebDAV 来实现上传功能。WebDAV (Web-based Distributed Authoring and Versioning) 是一种基于 HTTP 1.1 的通信协议，并在其基础之上添加了一些新的方法，使得应用程序可对 HTTP Server 直接进行读写，并支持文件锁和版本控制。

#### 11.1.1 基本构建

支持 HTTP Server 的软件有很多，我们就以最常见的 Apache 为例，在 station103 上给大家演示如何构建一个 WebDAV 文件共享服务。它的构建过程非常简单，只需三步即可完成。





## 1. 创建配置文件

Apache 默认支持 WebDAV，主配置文件 `/etc/http/conf/httpd.conf` 中应当已经包含了如下内容。`httpd` 服务在启动时会装载 WebDAV 模块及其相关文件。

```
LoadModule dav_module modules/mod_dav.so
LoadModule dav_fs_module modules/mod_dav_fs.so
<IfModule mod_dav_fs.c>
    # Location of the WebDAV lock database.
    DAVLockDB /var/lib/dav/lockdb
</IfModule>
```

编辑配置文件，在 221 行添加如下内容。

```
Include conf/mod_dav.conf
```

接下来，我们在 `/etc/http/conf/` 目录下创建一个名为 `mod_dav.conf` 的配置文件，在虚拟目录 `share` 上开启 WebDAV，并设置密码验证访问和 ACL。声明只有 192.168.0.83 这台主机，在正确地输入了用户名和密码的情况下，才可以实现对 `share` 目录的访问。

```
Alias /webdav "/var/www/html/share"
<Directory "/var/www/html/share">
    Dav On
    # Enable password authentication
    AuthType Basic
    AuthName DAV
    AuthUserFile /var/www/html/authfile
    require valid-user
    <LimitExcept GET PROPFIND OPTIONS>
        require valid-user
    </LimitExcept>
    # Access control list
    Order allow,deny
    Allow from 192.168.0.83
</Directory>
```

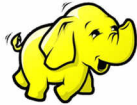
## 2. 创建共享目录

因为共享目录的空间需求很大，建议将它挂载到一个独立的分区上。假设这个分区的挂载点是 `/export/` 目录，我们在它的下面再创建一个名为 `share` 的子目录，然后在 `/var/www/html/` 目录下，给 `share` 创建一个符号链接。

```
mount a partition on /export/
# mkdir -p /export/share
# chown apache.apache /export/share
# ln -s /export/share /var/www/html/share
```

## 3. 创建用户密码

`AuthUserFile` 定义了用户密码文件的存放位置，我们使用 `htpasswd` 命令创建用户和密码。本例中的用户名是 `user`，密码为 `123`。



```
[root@station103 ~]# htpasswd -cm /var/www/html/authfile user
New password:
Re-type new password:
Adding password for user user
```

```
[root@station103 ~]# cat /var/www/html/authfile
user:$apr1$NVf/UknQ$8EHwGIhpn440F9wZ.Au8p.
```

#### 4. 客户端测试

重启 httpd 服务后 WebDAV 文件共享即可生效。在 redhat01 上使用 curl 测试是否可以匿名访问。预期结果应当是服务端返回 401 错误。

```
// 测试匿名访问
[root@redhat01 ~]# curl -L http://192.168.0.73/share
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Authorization Required</title>
</head><body>
<h1>Authorization Required</h1>
<p>This server could not verify that you
are authorized to access the document
requested. Either you supplied the wrong
credentials (e.g., bad password), or your
browser doesn't understand how to supply
the credentials required.</p>
<hr>
<address>Apache/2.2.15 (CentOS) Server at 192.168.0.73 Port 80</address>
</body></html>
```

接着，我们再来测试文件上传。redhat01 的 IP 地址是 192.168.0.83，ACL 的配置里只允许这台主机上传文件。注意：如果 WebDAV 服务器上已经存有和上传文件同名的文件，已有文件会被上传文件覆盖，并且没有任何提示。

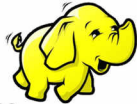
```
// 创建三个文件
[root@redhat01 ~]# touch {1,2,3}

// 测试上传单个文件
[root@redhat01 ~]# curl -u user:123 -T '1' http://192.168.0.73/share/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>201 Created</title>
</head><body>
<h1>Created</h1>
<p>Resource /share/1 has been created.</p>
<hr />
<address>Apache/2.2.15 (CentOS) Server at 192.168.0.73 Port 80</address>
</body></html>
```

```
// 测试上传多个文件
```







```
[root@redhat01 ~]# curl -u user:123 -T '{2,3}' http://192.168.0.73/share/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>201 Created</title>
</head><body>
<h1>Created</h1>
<p>Resource /share/2 has been created.</p>
<hr />
<address>Apache/2.2.15 (CentOS) Server at 192.168.0.73 Port 80</address>
</body></html>
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>201 Created</title>
</head><body>
<h1>Created</h1>
<p>Resource /share/3 has been created.</p>
<hr />
<address>Apache/2.2.15 (CentOS) Server at 192.168.0.73 Port 80</address>
</body></html>
```

此外，上传路径的结尾有没有“/”非常重要。如果有“/”，代表文件要上传到哪个目录。反之则代表文件上传后的文件名。本例中，如果 share 的后面没有加“/”，curl 会误认为是将文件放置到 Apache 的主目录 /var/www/html/ 下，并尝试将其重命名为 share。但是，主目录 /var/www/html/ 是不允许写入的。这样服务端就会返回 301 错误。

```
[root@redhat01 ~]# curl -u user:123 -T '1' http://192.168.0.73/share
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://192.168.0.73/share/">here</a>.</p>
<hr>
<address>Apache/2.2.15 (CentOS) Server at 192.168.0.73 Port 80</address>
</body></html>
```

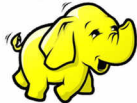
最后，我们在另外一台主机 redhat02 上测试是否可以上传文件。预期结果应当是服务端返回 403 错误。

### 1) 创建一个文件：

```
[root@redhat01 ~]# touch hello
```

### 2) 测试是否可以上传文件：

```
[root@redhat02 ~]# curl -u user:123 -T 'hello' http://192.168.0.73/share/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
```



```
<p>You don't have permission to access /share/hello
on this server.</p>
<hr>
<address>Apache/2.2.15 (CentOS) Server at 192.168.0.73 Port 80</address>
</body></html>
```

## 5. umask

系统初始时，文件权限和目录权限的默认值分别是 666 和 777。这样的权限配置形同虚设，毫无安全性可言。umask 可以被看作进一步实施权限裁剪的收敛方案。它采用相同置 0、不同保持的算法。经过 umask 叠加处理后的新权限会更合理、更安全。

在 Apache 中有两处可以设置 umask，它默认遵循全局配置文件 /etc/profile 中的设定。完成文件上传后，我们看到文件的权限是 644。

```
[root@station103 share]# ls -l
total 132
-rw-r--r-- 1 apache apache    0 Jun 30 15:31 1
-rw-r--r-- 1 apache apache    0 Jun 30 15:31 2
-rw-r--r-- 1 apache apache    0 Jun 30 15:31 3
```

/etc/profile 中定义了 UID 小于 200 的用户的 umask 是 022，所以 666 叠加 umask 后的值为 644。

```
[root@station103 shara]# id apache
uid=48(apache) gid=48(apache) groups=48(apache)

[root@station103 shara]# grep -C1 umask /etc/profile

# By default, we want umask to get set. This sets it for login shell
# Current threshold for system reserved uid/gids is 200
--
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 002
else
    umask 022
fi
```

Apache 也可以在 /etc/sysconfig/httpd 中定义自己的 umask。例如，我们希望用户只能上传文件，但上传后不能删除它们。那么，应当将 umask 的值设置成 222。

```
# echo "umask 222" >> /etc/sysconfig/httpd
```

## 11.1.2 WebDAV on HTTPS

虽然，我们为 WebDAV 设置了密码验证和 ACL，将访问者控制在特定用户、特定主机的范围内。但是，运行这样的 WebDAV 依旧存在一定的安全隐患。HTTP 是明文协议的，任何人都可以通过 Sniffer 工具去截获敏感信息。

例如，我们在 redhat01 这台主机上启用 tcpdump 侦听，并将结果保存到文件。只要借

助 Wireshark 进行简单的分析, 很容易就能找到用户名和密码, 如图 11-1 所示。为了防止嗅探攻击, 应当将 HTTP 协议替换成 HTTPS 协议。

```
# tcpdump -w webdav_over_http.cap

Hypertext Transfer Protocol
GET /share/1 HTTP/1.1\r\n
Authorization: Basic dXNlcjoxMjM=\r\n
Credentials: user:123
User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.14.0.0 zlib/1.2.3 libidn/1.18 libssh2/1.4.2
Host: 172.25.11.73\r\n
Accept: */*\r\n
Content-Length: 0\r\n
Expect: 100-continue\r\n
\r\n
```

图 11-1 使用 Wireshark 分析 HTTP 中包含的用户名和密码

### 1. 生成证书请求

在 Apache 服务器上, 生成密钥对, 并使用密钥生成一个证书请求。在填写证书请求的时候, 一定要注意 CN 项的名称应当与客户端访问时所用的名称保持一致。如果客户端要访问的是域名, CN 里面填写的也应当是域名; 如果客户端要访问的是 IP 地址, CN 对应填写的就应当是 IP 地址。

首先, 执行子命令 `genrsa` 生成密钥对。

```
[root@station103 ~]# openssl genrsa -out server.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
....+++
e is 65537 (0x10001)
```

然后, 使用子命令 `req` 生成证书请求。

```
[root@station103 ~]# openssl req -new -key server.key -out server.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:CN
State or Province Name (full name) []:Beijing
Locality Name (eg, city) [Default City]:Beijing
Organization Name (eg, company) [Default Company Ltd]:example.com
Organizational Unit Name (eg, section) []:Tech
Common Name (eg, your name or your server's hostname) []:192.168.0.73
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
```



```
An optional company name []:
```

## 2. 签发证书

在 CA 服务器上签发这张证书请求。符合规范的证书应当由可信的权威 CA 签发，而内部证书大都使用自签名证书。下面示例中所签发的就是一张自签名证书。

```
[root@station103 ~]# openssl x509 -req -days 3650 -in server.csr -signkey
server.key -out server.crt
Signature ok
subject=/C=CN/ST=Beijing/L=Beijing/O=example.com/OU=Tech/CN=192.168.0.73
Getting Private key
```

## 3. Apache 配置 SSL

最后，我们将刚刚生成的证书和私钥分别复制到目录 `/etc/pki/tls/certs/` 和 `/etc/pki/tls/private/` 下面。在确保系统已经安装 `mod_ssl` 软件包后，编辑文件 `/etc/http/conf.d/ssl.conf`，修改如下两行即可。修改完成后，重启 `httpd` 服务即可生效。

```
SSLCertificateFile /etc/pki/tls/certs/server.crt
SSLCertificateKeyFile /etc/pki/tls/private/server.key
```

## 4. 客户端测试

客户端执行如下命令进行测试。自签名证书缺乏可信的、完整的证书链，客户端在使用 `curl` 连接时会被拒绝。此时可使用选项 `-k` 来绕过验证，它表示不论何种情况，你都要强制使用 HTTPS 去访问这个站点。

```
# curl -k -u user:123 0-T '{1,2,3}' https://192.168.0.73/share/
```

# 11.2 构建 NFS 服务

在数据中心的应用区里，同类业务的应用程序之间会有一些共享数据。它们通常是一些文本、图片或者报表，不便存放在数据库中。应用程序对这些数据是实时访问的，对服务的可靠性和读写性能要求较高。这种场景适合部署 NFS Cluster。

NFS 是基于 RPC 的一个服务。RPC (Remote Procedure Call Protocol) 是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。它跨越了传输层和应用层，在通信程序之间携带信息数据，使基于网络的应用程序开发变得更加容易。

## 11.2.1 NFS v4 的新特性

NFS 的最新版本发展到了 v4。与之前的老版本相比，发生了很多新的变化。最为显著的变化就是 NFS v4 集成了锁机制。锁机制对于文件请求访问来说是非常重要的。如果没有锁，当多个客户端同时对一个文件进行 I/O 操作时，会引发该文件数据块的损坏。在 NFS v4 之前，服务需要借助 NLM (Network Lock Manager) 的协助实现锁的操作。而 NFS v4



自身已经实现了相应的功能，不再依靠外部支援了。这种集成方案使得各组件之间配合得更加紧密。由于这个功能的出现，NFS v4 对于 RPC 服务的依赖也降低了不少，已经不再使用 `rpc.statd` 和 `rpc.lockd`。

早期的 NFS v2 和 NFS v3 支持 UDP。UDP 的性能比 TCP 要好，而且无状态的 UDP 使得服务器可以毫无负担地重启。NFS v4 明确要求传输层必须支持拥塞控制，所以 UDP 也不再被使用了。

NFS 作为共享服务，原本就允许多个客户端并发访问。为了保持文件同步，客户端需要经常向服务器发起请求，去获取文件属性信息，以判断其他人是否修改了该文件。频繁地向服务器发送请求会降低系统的性能。NFS v4 使用了 `Delegation` 技术优化这个问题。当客户端访问一个文件时，系统会根据锁的情况，决定是否给客户端分配一个 `Delegation`。凡是持有 `Delegation` 的客户端，就代表它和文件之间是同步的。这样，客户端就没有必要反复向服务器发送确认同步的请求了。如果访问的文件已经被其他客户端执行写操作，`Delegation` 就不会被分配。

除此之外，NFS v4 还有以下一些新特性。

- ☐ 更高的安全性；
- ☐ 增强对 Windows 系统的兼容性；
- ☐ 允许自定义文件属性；
- ☐ 允许多个请求合并到一个 RPC 调用之中以降低延迟响应；
- ☐ 支持并行存储。

### 11.2.2 NFS 常见问题处理

本节将为大家介绍使用 NFS 服务时一些常见问题的处理方法。

#### 1. Squash

文件对象的所属角色有三种：`Owner`、`Group` 和 `Other`。文件系统借助 `UID` 和 `GID` 来判断用户角色，然后依照角色对应的权限决定用户如何去访问它。当使用 NFS 访问远程共享文件时，共享目录下的文件的 `UID` 和 `GID` 会被传递到本地的文件系统上。如果共享文件的 `UID` 在本地的文件系统上也有对应的用户，就会以这个用户的名字来显示，从而产生权限冒领的风险。别有用心的人可以借此非法获取授权。`Squash` 会将共享文件的 `UID` 和 `GID` 重定向到一个新值，这个值被系统预留作为 `nobody` 的角色。被设置为 `nobody` 之后，只有真正的 `Owner` 可以读写，其他人只能出任 `Other` 的角色。

我们在 `redhat03` 这台主机上配置了一个 NFS 服务，将目录 `/share/` 设置成共享。主机 `redhat01` 和 `redhat02` 使用 NFS v3 挂载这个共享目录。另外，三台主机分别创建了如下用户。

```
[root@redhat01 ~]# id luci && id test
uid=500(luci) gid=500(luci) groups=500(luci)
```

```
uid=1000(test) gid=1000(test) groups=1000(test)
```

```
[root@redhat02 ~]# id ricci && id test
uid=500(ricci) gid=500(ricci) groups=500(ricci)
uid=1000(test) gid=1000(test) groups=1000(test)
```

```
[root@redhat03 ~]# id se
uid=500(se) gid=500(se) groups=500(se)
```

三台主机均以 root 登录，在共享目录下分别创建了名为 01、02 和 03 三个文件。redhat01 以 test 登录创建了名为 test 的文件，redhat03 以 se 登录创建了名为 se 的文件。在主机 redhat03 上，使用 ls 命令查询文件属性，如下所示。

```
[root@redhat03 ~]# ls -l /share
total 16
-rw-r--r-- 1 nfsnobody nfsnobody 22 Jun 29 15:24 01
-rw-r--r-- 1 nfsnobody nfsnobody 7 Jun 29 15:23 02
-rw-r--r-- 1 root root 0 Jun 29 15:11 03
-rw-rw-rw- 1 se se 40 Jun 29 15:28 se
-rw-rw-rw- 1 1000 1000 9 Jun 29 15:30 test
```

// 加上 -n 后，显示的是 Owner 的 UID

```
[root@redhat03 ~]# ls -ln /share
total 16
-rw-r--r-- 1 65534 65534 22 Jun 29 15:24 01
-rw-r--r-- 1 65534 65534 7 Jun 29 15:23 02
-rw-r--r-- 1 0 0 0 Jun 29 15:11 03
-rw-rw-rw- 1 500 500 40 Jun 29 15:28 se
-rw-rw-rw- 1 1000 1000 9 Jun 29 15:30 test
```

NFS v3 的默认设置是 root\_squash，客户端上使用 root 创建的文件都已经被 squash 成 nobody 了，但是普通用户还是原来的 UID，文件 se 在 redhat03 上的 Owner 是 se，但到了 redhat01 和 redhat02 上就变成了 luci 和 ricci。这是由于三个用户的 UID 都是 500 的缘故。

//redhat01 的查询结果

```
[root@redhat01 ~]# ls -l /share
total 16
-rw-r--r-- 1 nfsnobody nfsnobody 22 Jun 29 15:24 01
-rw-r--r-- 1 nfsnobody nfsnobody 7 Jun 29 15:23 02
-rw-r--r-- 1 root root 0 Jun 29 15:11 03
-rw-rw-rw- 1 luci luci 40 Jun 29 15:28 se
-rw-rw-rw- 1 test test 9 Jun 29 15:30 test
```

//redhat02 的查询结果

```
[root@redhat02 ~]# ls -l /share
total 16
-rw-r--r-- 1 nfsnobody nfsnobody 22 Jun 29 15:24 01
-rw-r--r-- 1 nfsnobody nfsnobody 7 Jun 29 15:23 02
-rw-r--r-- 1 root root 0 Jun 29 15:11 03
-rw-rw-rw- 1 ricci ricci 40 Jun 29 15:28 se
```



```
-rw-rw-rw- 1 test      test      9   Jun 29 15:30 test
```

使用 NFS v4 挂载后，默认设置是 `all_squash`。除了本地 `root` 以外，都变成了 `nobody`。

```
[root@redhat03 ~]# ls -l /share
total 16
-rw-r--r-- 1 nfsnobody nfsnobody 22   Jun 29 15:24 01
-rw-r--r-- 1 nfsnobody nfsnobody  7   Jun 29 15:23 02
-rw-r--r-- 1 root      root      0    Jun 29 15:11 03
-rw-rw-rw- 1 nobody    nobody    40   Jun 29 15:28 se
-rw-rw-rw- 1 nobody    nobody     9    Jun 29 15:30 test
```

// 加上 `-n` 后，显示的是 Owner 的 UID

```
[root@redhat03 ~]# ls -ln /share
total 16
-rw-r--r-- 1 65534 65534 22   Jun 29 15:24 01
-rw-r--r-- 1 65534 65534  7   Jun 29 15:23 02
-rw-r--r-- 1 0      0      0    Jun 29 15:11 03
-rw-rw-rw- 1 99     99     40   Jun 29 15:28 se
-rw-rw-rw- 1 99     99     9    Jun 29 15:30 test
```

## 2. 重载配置

NFS 的共享配置文件位于 `/etc/exports`，如果同一个共享目录需要指定多个客户端挂载，可以按照如下格式书写。

```
[root@station101 ~]# cat /etc/exports
/export 192.168.0.87(rw,sync) \
        192.168.0.85(rw,sync) \
        192.168.0.83(ro,sync)
```

配置完成后不要通过重启 NFS 服务来加载新的配置项。这样做很危险，因为重启的过程中有可能会发生文件访问。另外，通常 NFS 服务是搭建在集群之上的，如果重启服务会被集群认为是节点故障，从而引发切换动作。如果此时切换失败，后果会非常严重。

命令 `exportfs -av` 用于加载那些新的配置条目。命令 `exportfs -uv` 会删除当前缓存中的配置条目，但不会真正删除 `/etc/exports` 配置文件中的内容。

```
[root@station101 ~]# exportfs -av
exporting 192.168.0.87:/export
exporting 192.168.0.85:/export
exporting 192.168.0.83:/export
```

```
[root@station101 ~]# exportfs -uv 192.168.0.83:/export
unexporting 192.168.0.83:/export
```

命令 `showmount -e` 用于查看 NFS 服务器提供的共享列表。

```
[root@redhat03 export]# showmount -e 192.168.0.71
Export list for 192.168.0.71:
192.168.0.83(rw,sync) *
/export                192.168.0.83,192.168.0.85,192.168.0.87
```

### 3. 减少连接数

如果客户端的数量很多，服务器上一直会保持比较大的连接数，对于资源的消耗会很大。而且客户端挂载后，未必一直都处于访问状态。使用 AutoFS 可以实现共享目录的自动挂载和卸载，只需要在客户端上安装软件包 `autofs`，并启动 `autofs` 服务即可。我们在 `station101` 这台主机上也配置了一个 NFS 服务，然后使用 `redhat03` 作为客户端，在上面部署了 `autofs`。

```
[root@redhat03 ~]# showmount -e 192.168.0.71
Export list for 192.168.0.71:
/export 192.168.0.87
[root@redhat03 ~]# mount 192.168.0.71:/export /mnt
```

我们先用普通的 `mount` 命令挂载 NFS，在 `station101` 上可以看到会话连接已经建立。只要你不卸载掉 `/mnt` 目录，这个会话连接会一直保持。

```
[root@station101 ~]# netstat -atnup |grep 2049
tcp  0  0  0.0.0.0:2049          0.0.0.0:*            LISTEN      -
tcp  0  0  192.168.0.71:2049    192.168.0.87:867     STABLISHED  -
tcp  0  0  :::2049              :::*                  LISTEN      -
udp  0  0  0.0.0.0:2049         0.0.0.0:*            -
udp  0  0  :::2049              :::*                  -
```

现在我们更换成 AutoFS 的方式再次登录测试。

```
// 登录时没有任何目录，登录后才会触发目录挂载，只有进入 export 后才会建立会话连接
[root@redhat03 ~]# cd /net
[root@redhat03 net]# ls
[root@redhat03 net]# cd 192.168.0.71
[root@redhat03 192.168.0.71]# ls
export
[root@redhat03 192.168.0.71]# cd export/
[root@redhat03 export]# ls
lost+found
// 退出目录 192.168.0.71 后，默认五分钟之后再次执行 ls 命令会发现目录已经被卸载掉了。
[root@redhat03 export]# cd ..
[root@redhat03 192.168.0.71]# cd ..
[root@redhat03 net]# ls
192.168.0.71
[root@redhat03 net]# ls
```

如果 NFS 服务挂了，使用 `mount` 命令挂载的客户端在访问共享挂载点时会 `hang` 死在那里，但是 AutoFS 则不需要有这样的担心，顶多就是告诉你找不到对象。

```
[root@redhat03 ~]# cd /net/
[root@redhat03 net]# ls
[root@redhat03 net]# cd 192.168.0.71
-bash: cd: 192.168.0.71: No such file or directory
```

AutoFS 默认超时时间是 5 分钟，只要返回到 `/net/` 目录下，`timeout` 计时就会开始，如果要调整这个时间，可在配置文件 `/etc/sysconfig/autofs` 中作如下修改。

```
[root@redhat03 ~]# grep ^TIMEOUT /etc/sysconfig/autofs
TIMEOUT=300
```

#### 4. mount 超时

NFS 磁盘和本地磁盘不是在同一时间挂载的，NFS 的挂载需要通过网络完成，因此它的挂载服务 netfs 位于 network 服务之后，这是由服务启动顺序决定的。

```
[root@redhat01 ~]# ls /etc/rc3.d/* |egrep "network|netfs"
/etc/rc3.d/S10network
/etc/rc3.d/S25netfs
//S 代表启动，后面的数字越小，启动的顺序越靠前。
```

当 netfs 启动开始挂载磁盘时，如果没有联系上 NFS 服务器，这个服务就会卡在那里久久不肯结束。客户端会非常执着地等待并重试这个操作。解决这个问题方法是：在 /etc/fstab 文件中设置如下两个挂载选项。

```
192.168.0.71:/export /mnt nfs defaults,retry=0,timeo=1 0 0
```

配置完成后，我们停止 NFS 服务再去尝试挂载，一秒钟之后系统会立即报错，不必再担心客户端 hang 死了。

```
[root@station103 ~]# mount -a
mount.nfs: Connection timed out
```

#### 5. 防火墙

NFS 是依赖于 RPC 服务的，默认情况下，rpcbind 的调用端口都是随机的。每重启一次 NFS 服务，nfs\_lock 等端口都会发生改变，如果客户端和 NFS 服务器之间有防火墙，则会受到影响。为此我们必须锁定这些端口，在配置文件 /etc/sysconfig/nfs 中，打开如下这些注释即可锁定相关端口。配置完成后，多次重启 NFS 服务，并使用 rpcinfo -p 进行检查，确定端口锁定成功。

```
[root@station103 ~]# grep 'PORT=' /etc/sysconfig/nfs
#RQUOTAD_PORT=875
#LOCKD_TCP_PORT=32803
#LOCKD_UDP_PORT=32769
#MOUNTD_PORT=892
#STATD_PORT=662
#STATD_OUTGOING_PORT=2020
#RDMA_PORT=20049
```

### 11.2.3 NFS 高可用方案

NFS 作为生产环境中主要的文件共享服务形式，确保它的高可用是首要考虑的事情。CentOS 自带集群部署套件 RHCS (RedHat Cluster Suit)，它的主要核心组件包括如下内容。

#### (1) CMAN

CMAN (Cluster Manager) 负责集群仲裁和成员资格管理。CMAN 通过监控集群节点



数目来了解集群仲裁。只有活跃节点过半时集群才具有仲裁资格。反之，集群活动将停止。当集群成员发生变化时，CMAN 会通知其他组件采取适当的行动。如果集群节点在规定的时间内没有传送消息，CMAN 会将它从集群中删除，并通知大家该节点已经不再是集群成员了。此时，其他组件会根据通知决定采取怎样的行动。

#### (2) 锁管理

在 RHCS 中，使用 DLM (Distributed Lock Manager) 作为锁管理器，它提供了一种同步集群节点对共享资源的访问机制。它分布于集群内的所有节点上，GFS2、CLVM 和 rgmanager 都使用它来同步状态。

#### (3) Fence

Fence 负责将故障节点从集群当中踢掉，切断 I/O 以确保数据的完整性。当 CMAN 通告某个节点已经不再是集群成员的时候，守护进程 fenced 将负责执行踢人任务。这时，DLM 和 GFS2 (如果配置了) 将暂停活动，直到它们检测到 fencing 执行完毕后，DLM 会释放对失败节点的锁定，GFS2 则恢复故障节点的日志。

#### (4) rgmanager

rgmanager (Resource Group Manager) 负责启动、停止系统服务的工作，从而完成 Failover 中服务和资源的切换与接管。

#### (5) CCS

CCS (Cluster Configuration System) 用于创建、修改和查看 cluster.conf 集群配置文件。

#### (6) Conga

RHCS 有三种配置方法：直接写配置文件、C/S 模式和 B/S 模式。Conga 是一个用于安装、配置和管理 RHCS 的综合用户界面。其中，组件 luci 提供了一个 Web 管理界面，而组件 ricci 作为远程 Agent 负责更新集群信息。

### 11.2.4 NFS Cluster 实施条件

在实施之前，请先确认生产环境中的前置条件是否已经准备完毕，是否符合部署 RHCS 的要求。

#### (1) 禁用 NetworkManager 服务

RHCS 不支持 NetworkManager 服务管理。如果服务器节点部署了图形化，这个服务通常是启用的，请使用 network 服务替换掉它。

#### (2) 禁用 acpi 服务

在部署了 RHCS 的集群节点上，请确认已经禁用了 acpi (Advanced Configuration and Power Management Interface) 服务。不同的 Fence 设备，其 Fence 效果也是不同的。有些 Fencing 等同于直接拔电源线，而有些则是发起一个正常的关机命令。后者的关机速度要比前者慢得多。请选择关机速度更快的方案作为你的 Fence 设备，例如 IPMI 就是一个很好的

选择。因为当需要 Fencing 时，这个节点对于外部而言就是不可用的，没必要按正常方式去关闭，而且正常关机也很可能根本就关不掉。

同样，acpi 服务如果在集群节点上是开启的，而且它参与了电源管理的活动，它会在重启主机时消耗相当长的时间。如果此时在关闭过程中出现了 Kernel Panic 或者因其他故障导致系统 hang 死而无法关闭节点，Fencing 将会失效，从而必须转入手工接管的模式。另外，有些服务器在夏季来临时经常莫名其妙地重启，这也和这个服务有关系。服务器内部有一个温度传感器，当它感知到温度超过预设阈值时，就会调用 acpi 尝试重启服务器。但是，大多数情况下，这个重启的动作都是不明智的，误判的情况会更多一些。除了虚拟机以外，所有的物理节点都不建议大家去安装这个服务。

### （3）配置 hosts 文件

集群配置大多采取主机名的书写方式，因此名称解析是非常重要的。尽管我们可以使用 DNS 来处理名称解析。但是在集群配置中，一定要把名称解析写入 hosts 文件。不要过度依赖外部服务而增加不必要的故障点。

### （4）多播和 IGMP 的支持

RHCS 的各组件之间使用多播地址进行沟通。因此网络必须启用多播并支持 IGMP 协议，否则节点无法加入集群，从而导致集群创建失败。

### （5）是否需要配置仲裁盘

默认集群当中的每个节点都拥有一张选票，选举获胜的条件是选票过半。假设你的集群中有三个节点，它们在选举中的得票结果是 2:1，那么落单的那台主机就会被 Fencing 掉。如果这个节点正在访问某个非常重要的资源，被强制 Fencing 后就会带来非常严重的后果。在这种情况下，为了保住该节点，需要配置仲裁盘作为选举的补充方案。仲裁盘可在特定条件下，增加某个节点的票数，确保它不会被其他选民刷掉。

注意：仲裁和仲裁盘是两回事。为了防止脑裂，仲裁是必需的。仲裁分为两种：以太网仲裁和仲裁盘仲裁。对于经由以太网的仲裁方案，是由节点投票的 50% 加 1 组合而成。而仲裁盘的仲裁方案，则是由用户来指定条件。

### （6）多路径管理

大多数硬件存储设备都会提供专门的多路径管理软件。如果存储没有提供，也可以使用系统内置的软件包 device-mapper-multipath。在 CentOS 6 中安装 multipath 后，在 /etc/ 目录下是找不到配置文件的。不要担心，它的示例文件位于 /usr/share/doc/device-mapper-multipath-0.4.9/multipath.conf。将这个示例文件直接复制到 /etc/ 目录下，再根据你的实际环境进行配置。

devnode\_blacklist 语句用于排除不需要进行聚合链接的设备，如果发现本地盘出现在聚合设备中，请将它们添加到 devnode\_blacklist 里面。

```
devnode_blacklist {
```

```

wwid 26353900f02796769
devnode "(ram|raw|loop|fd|md|dm-[sr|scd|st])[0-9]*"
devnode "^hd[a-z]"
}

```

配置完成后，启动 multipathd 服务进行磁盘扫描。命令 multipath -ll 用于查询聚合设备，如果发现扫描结果有误，可以使用 multipath -F 清除缓存，然后执行 multipath -v2 重新进行扫描。

### (7) 缺少存储设备

在没有共享存储的情况下，可以考虑使用 iSCSI 的替代方案，使用一台服务器充当存储设备。在这台服务器上安装软件包 scsi-target-utils，按照如下内容修改配置文件 /etc/tgt/targets.conf 后，启动 tgtd 服务即可。

```

<target iqn.2008-09.com.example:server.target11>
  initiator-address 192.168.0.83
  initiator-address 192.168.0.85
  backing-store /dev/sdb
</target>

```

在 NFS Server 上安装软件包 iscsi-initiator-utils，执行如下命令完成共享存储的挂载。如下所示，假设使用 IP 地址为 192.168.0.71 的主机 station101 作为共享存储，redhat01 使用 iscsiadm 命令来挂载共享磁盘。

```

[root@redhat01 ~]# iscsiadm -m discovery -t st -p 192.168.0.71
Starting iscsid: [ OK ]
192.168.0.71:3260,1 iqn.2008-09.com.example:server.target11
[root@redhat01 ~]# iscsiadm -m node -T iqn.2008-09.com.example:server.target11
-p 192.168.0.71 -l
Logging in to [iface: default, target: iqn.2008-09.com.example:server.target11,
portal: 192.168.0.71,3260] (multiple)
Login to [iface: default, target: iqn.2008-09.com.example:server.target11,
portal: 192.168.0.71,3260] successful.

```

## 11.2.5 NFS Cluster 的实施

本次我们使用 redhat01.example.com 和 redhat02.example.com 这两台主机构建一个 NFS Cluster 环境，使用 192.168.0.89 作为 Cluster IP。

### 1. 安装 Conga

我们在两个节点上，都部署好 Conga 的两个组件 luci 和 ricci，并启动服务。

```

# yum install -y luci ricci
# service luci start
# chkconfig luci on
# service ricci start
# chkconfig ricci on

```

在浏览器地址栏中，选择其中一个节点进行登录。



<https://192.168.0.73:8084/>

登录 Cluster 管理界面如图 11-2 所示。

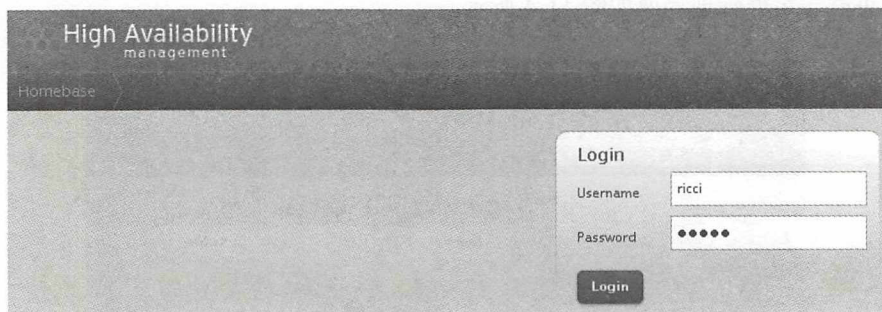


图 11-2 登录 Cluster 管理界面

这里可以使用任意系统用户验证登录。我们建议使用 ricci 用户进行管理，并将 ricci 用户密码设置成 root 的密码。

```
# usermod -p `awk -F: '/^root/ {print $2}' /etc/shadow` ricci
```

初始状态下，非 root 用户是没有授权的。你需要单击页面右上角的 Admin 链接进行授权。为 ricci 用户授权，如图 11-3 所示。

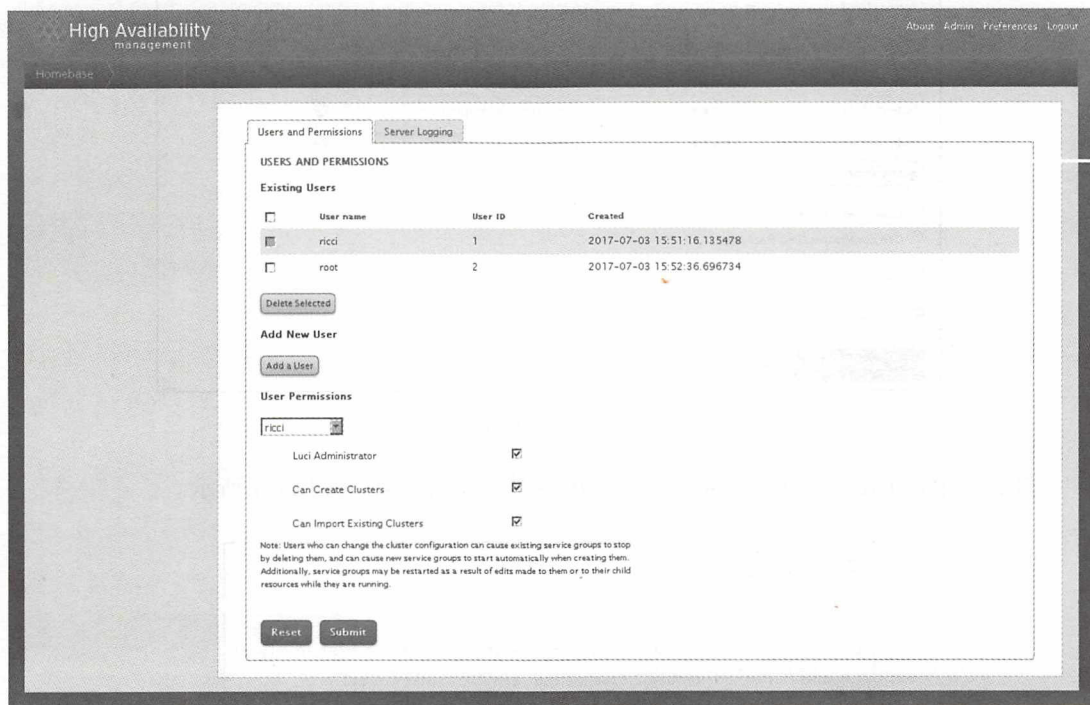


图 11-3 为 ricci 用户授权

## 2. 创建集群

登录成功后，单击 Manage Clusters 链接跳转到集群创建页面。再次单击 Create 按钮来创建一个集群。集群管理界面如图 11-4 所示。

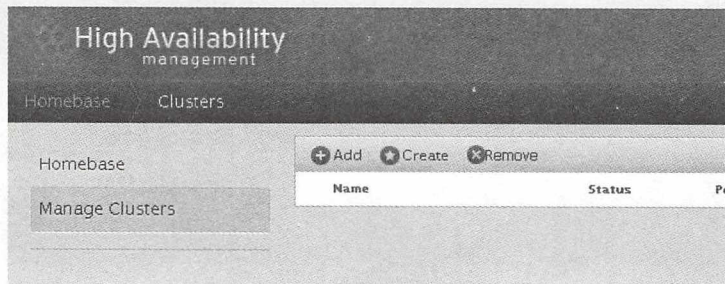


图 11-4 集群管理界面

创建集群界面如图 11-5 所示。系统对集群名字的长度是有要求的，不能超过 15 个字符。这里的密码指的是 ricci 的密码。节点连接成功后，ricci 会负责同步最终的配置。软件安装请选择本地安装，如果选择下载安装则可能会更新现有的组件版本。

| Node Name            | Password | Ricci Hostname       | Ricci Port |
|----------------------|----------|----------------------|------------|
| redhat01.example.com | •••••    | redhat01.example.com | 11111      |
| redhat02.example.com | •••••    | redhat01.example.com | 11111      |

图 11-5 创建集群

最后，单击 Create Cluster 按钮开始创建集群，如图 11-6 和图 11-7 所示。

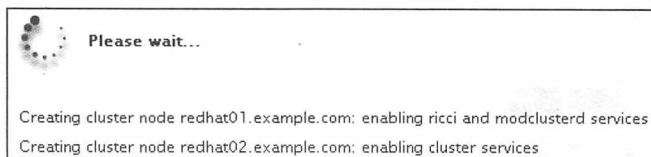


图 11-6 开始创建集群

| Nodes Fence Devices Failover Domains Resources Service Groups Configure  |                      |         |       |                |             |                      |
|--|----------------------|---------|-------|----------------|-------------|----------------------|
| <a href="#">Add</a> <a href="#">Reboot</a> <a href="#">Join Cluster</a> <a href="#">Leave Cluster</a> <a href="#">Delete</a> |                      |         |       |                |             |                      |
|  | Node Name            | Node ID | Votes | Status         | Uptime      | Hostname             |
| <input type="checkbox"/>   | redhat01.example.com | 1       | 1     | Cluster Member | 05:23:09:34 | redhat01.example.com |
| <input type="checkbox"/>   | redhat02.example.com | 2       | 1     | Cluster Member | 11:22:44:15 | redhat02.example.com |
| Select an item to view details   |                      |         |       |                |             |                      |

图 11-7 集群创建完毕

创建完成后，如果需要进一步调整集群的全局属性，请单击页面顶端的 Configure 链接，如图 11-8 所示。

|  |               |                  |           |                |           |
|--|---------------|------------------|-----------|----------------|-----------|
| Nodes  | Fence Devices | Failover Domains | Resources | Service Groups | Configure |
| <div> <a href="#">General</a> <a href="#">Fence Daemon</a> <a href="#">Network</a> <a href="#">Redundant Ring</a> <a href="#">QDisk</a> <a href="#">Logging</a> </div>             |               |                  |           |                |           |
| <h3>General Properties</h3> <p>Cluster Name <input type="text" value="NFS Cluster"/></p> <p>Configuration Version <input type="text" value="16"/></p> <p><a href="#">Apply</a></p> |               |                  |           |                |           |

图 11-8 配置集群全局属性

### 3. 配置 Fence Devices

正如前面所说，请你尽量选择一种可靠而快速的 Fence 方案，如图 11-9 所示。比较常用的是 IPMI Lan 或者 Dell iDRAC、HP iLO 等服务器专用接口。

|   |               |                  |           |                |           |
|---|---------------|------------------|-----------|----------------|-----------|
| Nodes   | Fence Devices | Failover Domains | Resources | Service Groups | Configure |
| <div> <a href="#">Add</a> <a href="#">Delete</a> </div>   |               |                  |           |                |           |
| <div> <div>Name</div> <div>Add Fence Device (Instance)</div> <div>-- Select a Fence Device --</div> <div> <a href="#">Submit</a> <a href="#">Cancel</a> </div> </div> |               |                  |           |                |           |

图 11-9 选择 Fence Device



#### 4. 创建故障域

如何去理解故障域呢？你可以将它看作集群内的一个分组。故障域的作用主要体现在多节点、多集群应用的场景之中。例如，我们有 A、B、C 三台主机。节点 A 和节点 B 分别部署了 Apache 和 NFS，节点 C 则同时部署了 Apache 和 NFS。我们将节点 A 和节点 C 加入一个故障域 Domain A，再将节点 B 和节点 C 加入另外一个故障域 Domain N。此时，节点 C 在 Domain A 中充当了节点 A 的备机，而在 Domain N 中则充当了节点 B 的备机。这就是故障域的作用。

实际上，这里的集群是一个很大的概念。一个集群不代表它只能有一个集群应用。划分集群应用的单位是故障域。一个集群内至少应当有一个故障域，它是用来对应后面要提到的服务组的。

如图 11-10 所示，Prioritized 是指系统在切换时会遵照优先级顺序执行。Restricted 限定了只有特定成员才可以切换。否则，切换任务可以在任意成员间进行。

**Add Failover Domain to Cluster**

Name:

☒ Prioritized      Order the nodes to which services failover.

☒ Restricted      Service can run only on nodes specified.

☐ No Failback      Do not send service back to 1st priority node when it becomes available again.

|  | Member               | Priority                              |
|--|----------------------|---------------------------------------|
|  | redhat01.example.com | <input checked="" type="checkbox"/> 1 |
|  | redhat02.example.com | <input checked="" type="checkbox"/> 2 |

图 11-10 创建故障域

#### 5. 创建资源

一个集群应用由很多元素组成，比如集群地址、服务、存储等。这些元素在 RHCS 中叫作资源。创建一个 NFS Cluster 至少应当配置四项资源。

首先，我们需要创建一个共享存储的资源，它就是集群后端挂载的共享存储设备。这里可以指定挂载设备、挂载点、文件系统、挂载参数等，如图 11-11 所示。

然后，我们创建 NFS Export 和 NFS Client，也就是 /etc/exports 配置文件的内容，如图 11-12 和图 11-13 所示。

再来创建 Cluster IP，如图 11-14 所示。

**Add Resource to Cluster**

Filesystem

**Filesystem**

Name: NFSDisk

Filesystem Type: ext4

Mount Point: /nfs

Device, FS Label, or UUID: /dev/sda

Mount Options:

Filesystem ID (optional):

Force Unmount: ☒

Force fsck: ☒

Enable NFS daemon and lockd workaround: ☒

Use Quick Status Checks: ☒

Reboot Host Node if Unmount Fails: ☒

Submit Cancel

图 11-11 创建资源之文件系统

**Add Resource to Cluster**

NFS v3 Export

**NFS v3 Export**

Name: NFSExport

Submit Cancel

图 11-12 创建资源之 NFS Export

**Add Resource to Cluster**

NFS Client

**NFS Client**

Name: NFSClient

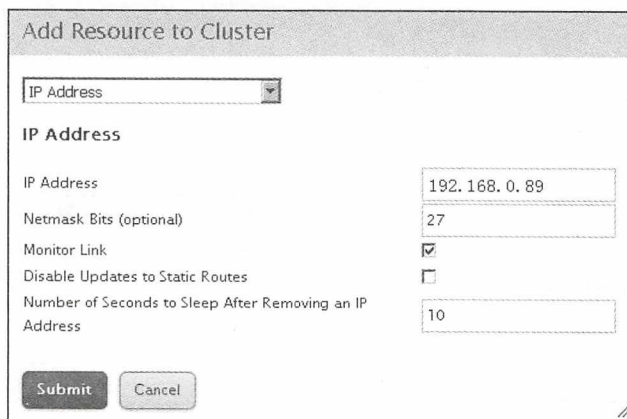
Target Hostname, Wildcard, or Netgroup: \*

Allow Recovery of This NFS Client: ☐

Options: rw,sync

Submit Cancel

图 11-13 创建资源之 NFS Client

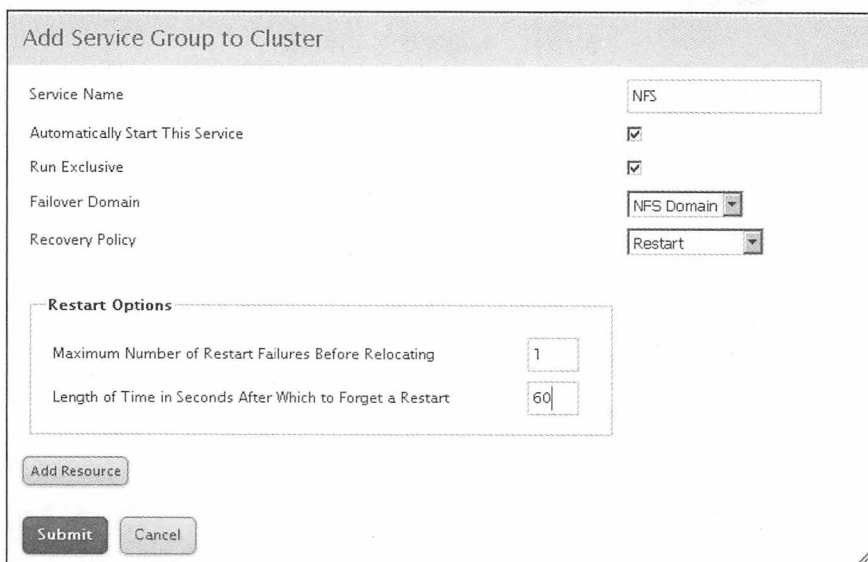


The dialog box titled "Add Resource to Cluster" contains a dropdown menu labeled "IP Address". Below it, the "IP Address" section includes several fields: "IP Address" with the value "192.168.0.89", "Netmask Bits (optional)" with the value "27", "Monitor Link" with a checked checkbox, "Disable Updates to Static Routes" with an unchecked checkbox, and "Number of Seconds to Sleep After Removing an IP Address" with the value "10". At the bottom are "Submit" and "Cancel" buttons.

图 11-14 创建资源之 Cluster IP

## 6. 创建服务并添加资源

最后一步，创建服务组并添加资源，如图 11-15 所示。



The dialog box titled "Add Service Group to Cluster" contains several fields: "Service Name" with the value "NFS", "Automatically Start This Service" with a checked checkbox, "Run Exclusive" with a checked checkbox, "Failover Domain" with a dropdown menu showing "NFS Domain", and "Recovery Policy" with a dropdown menu showing "Restart". Below these is a "Restart Options" section with two fields: "Maximum Number of Restart Failures Before Relocating" with the value "1" and "Length of Time in Seconds After Which to Forget a Restart" with the value "60". At the bottom are "Add Resource", "Submit", and "Cancel" buttons.

图 11-15 创建服务组并添加资源

单击 Add Resource 按钮，把刚才创建的四个资源 Filesystem、NFS Export、NFS Client 和 IP Address 依次添加到服务组中。请注意，NFS Client 是包含在 NFS Export 里面的，所以在添加 NFS Client 的时候，应单击位于 NFS Export 下方的 Add Child Resource 按钮。集群服务启动时，会依次加载各项资源。

集群创建完毕后可以看见，它们之间的关系是 Resource → Service Group → Failover Domain → Cluster。添加资源时，要注意各个资源之间的上下级关系，以及它们的启动顺序。



别忘了到另外一个节点上登录 `luci` 检查，如果看不到已经创建好的集群，需要单击 `Add` 按钮重新添加，确保两个管理端同步，如图 11-16 所示。

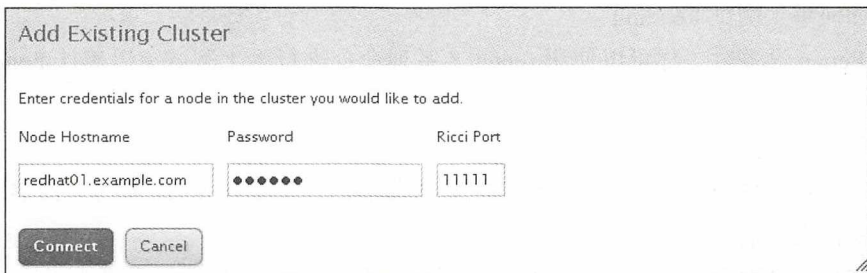


图 11-16 添加节点到已有集群

到这里，NFS Cluster 已经创建完毕了。集群服务启动后，我们可以使用命令 `clustat -i` 实时查看集群的状态。

```
[root@redhat01 ~]# clustat -i
Cluster Status for NFS Cluster @ Wed Jun 28 10:31:15 2017
Member Status: Quorate
```

| Member Name          | ID | Status                   |
|----------------------|----|--------------------------|
| redhat01.example.com | 1  | Online, Local, rgmanager |
| redhat02.example.com | 2  | Online, rgmanager        |

| Service Name | Owner (Last)         | State   |
|--------------|----------------------|---------|
| service:NFS  | redhat01.example.com | started |

### 11.2.6 NFS Cluster 故障排错

由于集群创建的前置条件和操作步骤比较多，所以在部署过程中，难免出现各种各样的问题，下面将一些常见故障的排错方法做个汇总。这里只涉及和集群配置相关的方面，服务器和存储等硬件问题请读者自行处理。

#### (1) 创建集群失败

集群在创建时，需要验证是否可以连接远程节点。验证失败是由于用户不可用、密码不符、权限不足或者无法完成名称解析造成的。你可以检查一下上述问题以确认故障原因。

另外，如果网络不支持多播和 IGMP，集群节点是无法相互通信的。请你检查网络环境是否支持，并确保 `NetworkManager` 服务已经关闭。

#### (2) 无法添加或删除节点

这种错误多半是由于节点之间的信息不同步造成的，可使用 `css` 命令检查各节点间的同步情况，或者直接执行 `css_sync` 命令重新发起同步。

```
// 以下两条命令执行任意一条即可  
# ccs -h <HOST> --checkconf  
# ccs_sync <HOST1> <HOST2> ...
```

### (3) 如何手工测试 Fencing

添加 Fence 设备时，你可以使用 fence\_\* 系列命令进行手工测试，以确认 Fencing 是正常的。

### (4) 仲裁盘未生效

如果你配置了仲裁盘，却发现无法将它加入集群内，请确保 qdisk 服务已经启动。

### (5) 集群无法启动

服务组所使用的资源来自于 /usr/share/cluster/ 目录下面的脚本，命令 rg\_test 会将执行这些脚本时的错误结果返回，可以使用如下方法来找出启动失败的原因。

```
# rg_test test /etc/cluster/cluster.conf start service <SERVICE_NAME>
```

### (6) 出现脑裂

如果每个节点上都表明自己是 online 而对方是 offline，这就是典型的脑裂表现。在没有仲裁盘的情况下，仲裁是经由以太网实现的，这时出现脑裂是由于节点无法通过心跳网络完成多播通信所导致的。请确认网络环境和链路状态是否正常，尤其是多播和 IGMP。

## 11.3 构建 SFTP 服务

还有一些文件共享来自于企业外部的数据交换，例如金融服务和银行对账时需要上传一些报表与证照。它们的应用是从外网连进来的，非内部人员访问，而且使用频率较低。因此，它对服务可靠性和数据备份的要求不是很高，但对于身份验证和数据安全传输却十分敏感。这种场景需要使用 SFTP 来实现。

### 11.3.1 Chroot SFTP 和公钥访问的必要性

SFTP 的应用场景是外网连入，这里我们暂不考虑网络层的防护，只从系统层面上考虑安全问题。虽然在创建 SFTP 用户时，我们会禁止它登录 Shell，但 SFTP Server 的本地用户还是可以登录的。如果本地用户的密码遭到泄露，这台 SFTP Server 就十分危险了。本地用户是按照角色划分的，而生产系统的用户密码又都是统一的。这样一来，SFTP Server 不就成了入侵者的跳板机了么？采用 Chroot SFTP 会彻底斩断入侵者的念想，而公钥访问还可以进一步加固安全。密码验证无法识别登录者的身份，入侵者在窃取密码后可以在任何一台主机上登录。公钥验证则锁定了登录设备，降低了非法访问的可能性。即便那台登录设备被入侵者控制，他登录到 SFTP Server 时，也会被锁定在 Chroot 环境里面，服务器本地系统则不会受到威胁。

公钥验证配置很简单, 在客户端执行 `ssh-keygen` 命令一路回车即可。然后将用户家目录下的 `.ssh/id_rsa.pub` 这个公钥文件发送给 SFTP Server 的管理员。管理员将其内容添加到 SFTP 用户家目录下的 `.ssh/authorized_keys` 文件中。

至此, 公钥验证配置完成, 下面介绍如何构建 Chroot SFTP 环境。

### 11.3.2 构建 Chroot SFTP

构建 Chroot SFTP 环境, 需要借助工具 `febootstrap` 来完成。这里的 `fe` 指的就是 Fedora。我们在 `redhat03` 上打造一个极简环境, 只包含必要组件, 将 `yum` 源指向 `192.168.0.70` 这台主机。

```
# febootstrap -i coreutils -i expect -i openssh-server centos6 chroot http://
192.168.0.70/centos6.5-x86_64
```

由于手工构建和维护 Chroot SFTP 很麻烦, 笔者专门写了一套脚本用于环境部署和后续的用户创建。这套脚本包括 `deploy`、`addkey`、`remove`、`startup`、`sftp` 和 `umount`。前三个脚本用于环境部署、添加 SFTP 用户以及整个环境的删除, 后面三个脚本分别对应前面三个脚本执行操作时的调用。

具体的使用方法如下所示。

```
# sh deploy.sh 部署 chroot SFTP, 指定部署目录为 /export/chroot
# sh addkey.sh <USER> <PUB_KEY> 添加一个 SFTP 账户
# sh remove.sh 会删除整个环境及数据, 已经上线的环境慎用
# chroot /export/chroot sh startup.sh 如果服务器重启, 开机后需要执行这个脚本
```

下面是各个脚本的源代码。

`depoly.sh` 的源代码如下所示, 它负责部署 `chroot` 环境以及其他脚本文件, 并完成 `fstab`、`sshd_config` 等配置工作。

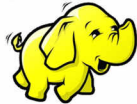
```
[root@redhat03 ~]# cat depoly.sh
#!/bin/bash

#Define Environment Variable
DEPOLYMENT_ROOT=/export
CHROOT=$DEPOLYMENT_ROOT/chroot

test -d /export/chroot && echo "$CHROOT is not create." && exit 2

function f_deploy ()
{
    mkdir -p $DEPOLYMENT_ROOT
    cp -rf ./chroot $CHROOT
    cp -f ./startup.sh $CHROOT
    cp -f ./sftp.sh $CHROOT
    cp -f ./umount.sh $CHROOT
```





```
cp -f /proc/cmdline $CHROOT/proc
rm -f $CHROOT/etc/localtime
cp -f /usr/share/zoneinfo/Asia/Shanghai $CHROOT/etc/localtime
egrep "tmpfs|devpts|sysfs|proc" /etc/fstab > $CHROOT/etc/fstab
cat > $CHROOT/etc/ssh/sshd_config << EOF
Protocol 2
Port 10022
PasswordAuthentication no
GSSAPIAuthentication no
UseDNS no
ClientAliveCountMax 5
ClientAliveInterval 120
AcceptEnv LANG LC_CTYPE LC_NUMERIC LC_TIME LC_COLLATE LC_MONETARY LC_MESSAGES
AcceptEnv LC_PAPER LC_NAME LC_ADDRESS LC_TELEPHONE LC_MEASUREMENT
AcceptEnv LC_IDENTIFICATION LC_ALL LANGUAGE
AcceptEnv XMODIFIERS
Subsystem sftp internal-sftp
ChrootDirectory %h
EOF
}

#Main Functions
f_deploy
chroot $CHROOT sh startup.sh
```

startup.sh 的源代码如下所示，它负责启动 sshd 进程，第一次启动会生成相关的密钥。当端口被占用时，会导致启动失败，需要先杀掉对应的进程。

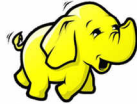
```
[root@redhat03 ~]# cat startup.sh
#!/bin/bash

mount -a
/etc/init.d/sshd start
PORT=`awk '/Port/ { print $NF }' /etc/ssh/sshd_config`
for i in `netstat -A inet -antup |awk -v var=$PORT ' $0 ~ var {split($NF,
    a,"/");print a[1]}`
do
    kill -9 $i
    echo kill process $i
done
/etc/init.d/sshd start
/etc/init.d/sshd status
```

addkey 的源代码如下所示，它将公钥文件复制到临时目录下，然后通过调用 sftp.sh 来完成账户的添加工作。

```
[root@redhat03 ~]# cat addkey.sh
#!/bin/bash

#Define Environment Variable
```



```
DEPOLYMENT_ROOT=/export
CHROOT=$DEPOLYMENT_ROOT/chroot
USER=$1
FILE=$2

if [ -z "$1" ] || [ -z "$2" ]; then
    echo "args is not enough."
    exit 2
fi

cp -f $FILE $CHROOT/tmp.pub
chroot $CHROOT sh sftp.sh "$USER"
```

sftp.sh 的源代码如下所示，它是添加 sftp 账户的核心脚本，完成一系列账户创建、权限分配等操作。

```
[root@redhat03 ~]# cat sftp.sh
#!/bin/bash

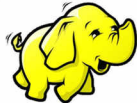
#Define Environment Variable
USER=$1
GROUP=$USER
HOME=/home/$USER
HOST=`ip addr show eth0 |awk -F " |/" '{print $6}'`
PORT=`awk '/Port/ { print $NF }' /etc/ssh/sshd_config`

id $USER &> /dev/null
if [ $? -ne 0 ]; then
    useradd -s /sbin/nologin $USER
    usermod -p `mkpasswd -l 50 -s 0` $USER
    mkdir $HOME/etc
    cp /usr/share/zoneinfo/Asia/Shanghai $HOME/etc/localtime
fi

mkdir -p $HOME/{.ssh,upload}
chown root:root $HOME
chmod 755 $HOME
chown $USER:$GROUP $HOME/.ssh
chmod 500 $HOME/.ssh
chown $USER:$GROUP $HOME/upload
chmod 755 $HOME/upload
cat tmp.pub >> $HOME/.ssh/authorized_keys
chown $USER:$GROUP $HOME/.ssh/authorized_keys
chmod 400 $HOME/.ssh/authorized_keys

echo "-----"
echo "OK. SFTP Account have built. Please testing."
echo "sftp -oPort=$PORT $USER@$HOST"
echo "-----"
```

remove.sh 的源代码如下所示，它会连同整个 sftp 环境和数据一并删除。请注意备份相



关数据。

```
[root@redhat03 ~]# cat remove.sh
#!/bin/bash

#Define Environment Variable
DEPOLYMENT_ROOT=/export
CHROOT=$DEPOLYMENT_ROOT/chroot

function f_check ()
{
    read -p "Dangerous! You will erase all data which users' uploaded! (n/y)" STRING
    if [ $STRING == "y" ]; then
        echo "Delete start....."
    else
        echo "Opearation is cancelled!"
        exit 1
    fi
}

#Main Functions
f_check
chroot $CHROOT sh umount.sh
rm -rf $CHROOT
```

umount.sh 的源代码如下所示，它是配合 remove 操作的。在操作之前，需要 umount 所有的挂载点。

```
[root@redhat03 ~]# cat umount.sh
#!/bin/bash

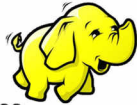
for i in `mount|awk '{print $1}`
do
    umount -l $i
done
```

我们使用这套脚本在 redhat03 上部署了一套 Chroot SFTP 环境。使用 redhat01 作为客户端进行测试。这套 SFTP 使用的端口是 10022，其目的就是，不让入侵者在第一时间轻易猜出这个端口后面真正部署的服务是什么。

```
// 创建一个空文件用于上传测试
[root@redhat01 ~]# touch file
[root@redhat01 ~]# chmod 777 file
[root@redhat01 ~]# ls -l file
-rwxrwxrwx 1 root root 0 Jun 29 09:29 file

// 登录到 redhat03 上测试 chroot 和 upload
[root@redhat01 ~]# sftp -oPort=10022 alice@192.168.0.87
Connecting to 192.168.0.87...
sftp> pwd
```





```

Remote working directory: /
sftp> ls
upload
sftp> cd ../../
sftp> pwd
Remote working directory: /
sftp> ls
upload
sftp> cd upload
sftp> put file
Uploading file to /upload/file
file                               100%  0    0.0KB/s   00:00
sftp> ls -l
-rwxr-xr-x  1 500      500              0 Jun 29 02:57 file

```

在测试的过程中我们发现了一个问题。我们在 09:29 创建了测试文件，上传文件的时间是 10:57，可这里执行 `ls -l` 显示的上传时间居然是凌晨 02:57，看来是时区有问题。但接下来更加奇怪的是，`ls -l file` 显示的上传时间又是对的。

注意：命令 `lls` 用于查看客户端本地文件。

```

[root@redhat01 ~]# sftp -oPort=10022 alice@192.168.0.87
Connecting to 192.168.0.87...
sftp> cd upload
sftp> ls -l
-rwxr-xr-x  1 500      500              0 Jun 29 02:57 file
sftp> ls -l file
-rwxr-xr-x  0 500      500              0 Jun 29 10:57 file
sftp> lls -l file
-rwxrwxrwx 1 root root 0 Jun 29 09:29 file

```

造成时区错误的原因是这样的。`ls -l` 不带参数返回的是服务器端的时区信息和文件属性。由于 `chroot SFTP` 将根目录限制在了 `SFTP` 用户的家目录下面，而系统的时区信息是存储在 `/etc/localtime` 上的，所以在无法访问 `/etc/localtime` 的情况下，系统以 UTC 0 的时区信息返回结果。而 `ls -l` 后面有文件对象的话，查询到的 `stat` 信息会在客户端上拼装，使用的是客户端的时区，但时间还是上传到服务器的时间。我们使用 `lls` 查询本地文件的创建时间是 09:29 而非 10:57，10:57 是上传到服务器的时间。我们进入 `chroot` 环境中，再次验证这一点。

```

[root@redhat03 ~]# chroot /export/chroot/
bash-4.1# ls -l /home/alice/upload/file
-rwxr-xr-x 1 alice alice 0 Jun 29 10:57 /home/alice/upload/file

```

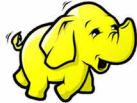
同时，我们还发现了另外一个现象。使用 `ls -l file` 查询时，它的 `inode` 数从 1 变成了 0，这也说明它的 `stat` 信息是在客户端上拼装的。由于这个 `stat` 信息在客户端上没有对应，所以 `inode` 数为 0。

解决这个问题的方法是在 `chroot SFTP` 家目录下面再构造一个 `localtime`。

```

[root@redhat03 ~]# chroot /export/chroot/

```



```

bash-4.1# mkdir /home/alice/etc
bash-4.1# cp /usr/share/zoneinfo/Asia/Shanghai /home/alice/etc/localtime
bash-4.1# ls -l /home/alice/
total 8
drwxr-xr-x 2 root root 4096 Jun 29 11:12 etc
drwxr-xr-x 2 alice alice 4096 Jun 29 10:57 upload
bash-4.1# ls -l /home/alice/etc/
total 4
-rw-r--r-- 1 root root 405 Jun 29 11:12 localtime

```

完成后我们再来验证，时间显示已经正常了。顺便说一下，我们的脚本 `sftp.sh` 已经将这个问题修正过了。如果读者想要测试，需要把 SFTP 用户家目录下的 `localtime` 文件删除掉。

```

[root@redhat01 ~]# sftp -oPort=10022 alice@192.168.0.87
Connecting to 192.168.0.87...
sftp> cd upload
sftp> ls -l
-rwxr-xr-x 1 500 500 0 Jun 29 10:57 file

```

如果我们希望为 SFTP 设置独立的 `umask`，只需修改配置文件 `sshd_config` 中的如下代码即可。

```

bash-4.1# cat /etc/ssh/sshd_config
Subsystem sftp internal-sftp -u 022

```

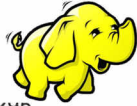
### 11.3.3 SFTP 容灾方案

由于 SFTP 服务不适合做集群，如果 SFTP 服务器宕机，如何快速恢复服务和数据呢？我们可以使用 `rsync` 同步作为 SFTP 服务器的容灾方案。因为 SFTP 本来就是对外提供服务的，它的场景特点相对于 NFS 服务来说使用频率较低，SLA 要求也不会太高，因此，它比较适合 `rsync` 容灾的特点。

#### 注意：

备份和容灾是两个完全不同的概念。备份的数据是静态的，数据可以是一份全备，也可以是一份全备加多份增量。而同步的数据是动态的，它会周期性地与生产端进行同步更新。如果在生产端上误删了数据，容灾端在随后一段时间内，同样也会将数据删除掉。容灾的目的是当生产端宕机无法对外提供服务和数据时，启动容灾端，恢复服务和绝大部分数据。虽然容灾的恢复效果弱于集群服务，但是它可以实现异地准冗余的手段，对于不适用于集群构建的 SFTP 服务来说是可行的。

构建这套 `rsync` 容灾体系的思路是这样来考虑的。因为创建 Chroot SFTP 服务时我们要挂载当前主机的 `/proc` 目录，`proc` 的内容是服务器当前的运行状态，这块是不能直接同步的。我们需要同步的只有数据、用户信息、用户家目录结构以及对应的权限。我们可以使用前面提供的部署脚本，在容灾端上直接创建一个新的 Chroot SFTP 服务环境，然后同步上述



内容即可。具体的实施步骤如下。

### 1. 容灾端

首先，我们使用 IP 地址为 192.168.0.85 的主机 redhat02 作为容灾服务器，执行 `deploy.sh` 创建一个新的 Chroot SFTP 服务环境。

然后，安装软件包 `xinetd` 和 `rsync`，并创建如下配置文件 `/etc/rsyncd.conf`。其中，每一个方括号代表一个同步模块，`path` 是指容灾端的同步目录的位置，`auth users` 是指同步时用于验证的用户，`secrets file` 是指用户密码文件的存储位置。这个配置文件包含了两个同步内容，因此需要两个同步模块。一个是同步家目录，另一个是同步用户、组和密码文件。

```
uid = root
gid = root
use chroot = no
max connections = 4
strict modes = yes
port = 873

[backup]
path = /export/chroot/home/
comment = This is a example
ignore errors
read only = false
list = no
hosts allow = 192.168.0.87
hosts deny = 0.0.0.0/0
auth users = rsync_user
secrets file = /etc/rsyncd.pw
pid file = /var/run/rsyncd.pid
lock file = /var/run/rsync.lock
log file = /var/log/rsyncd.log

[account]
path = /export/chroot/etc/
comment = This is a example
ignore errors
read only = false
list = no
hosts allow = 192.168.0.87
hosts deny = 0.0.0.0/0
auth users = rsync_user
secrets file = /etc/rsyncd.pw
pid file = /var/run/rsyncd.pid
lock file = /var/run/rsync.lock
log file = /var/log/rsyncd.log
```

最后，使用 `mkpasswd` 命令生成一个随机密码文件 `/etc/rsyncd.pw`，并修改权限。

```
# echo "rsync_user:`mkpasswd -l 10 -s 0`" > /etc/rsyncd.pw
```



```
# chmod 600 /etc/rsyncd.pw
```

## 2. 生产端

首先，将在 redhat02 上创建的密码文件复制到 redhat03 上，完成编辑后调整权限。注意：redhat02 上的密码文件是包含用户名和密码的，而 redhat03 上的密码文件只有密码，这两者是有区别的。

```
# scp 192.168.0.85:/etc/rsyncd.pw /etc/rsyncd.pw
# sed -i "s/rsync_user://" /etc/rsyncd.pw
# chmod 600 /etc/rsyncd.pw
```

然后设定 crontab 任务，让 redhat03 周期性地执行如下两条命令就可以了。

```
# rsync -vzrtopg --delete --force --progress --password-file=/etc/rsyncd.pw /
export/chroot/home/ rsync_user@192.168.0.85::backup

# rsync -vzrtopg -delete --force --progress --password-file=/etc/rsyncd.pw /
export/chroot/etc/passwd /export/chroot/etc/shadow /export/chroot/etc/group
rsync_user@192.168.0.85::account
```

## 3. 用户登录容灾服务器失败

这套系统构建完成后，我们可以在客户端上使用 sftp 命令去测试。如果发现用户登录容灾服务器失败，无外乎就是用户信息、权限或者配置不同所造成的问题。排错时可以从如下这几个方面来入手。

首先，请你确定容灾端的目录结构是否正确。

在书写同步命令的时候，依旧要格外注意源文件路径结尾的“/”的含义。因为容灾服务器创建完 Chroot SFTP 环境后，已经包含了 home 目录。执行同步时，我们只要它下面的内容，因此必须在结尾处加上“/”。如果没有这个“/”，在同步时会连同 home 目录本身都复制过去，这样容灾端的 home 目录下就会再多出一个 home 目录，这样肯定会出错。

其次，UID 和 GID 串号导致权限不匹配。我们来看一个例子，在生产端 redhat03 上，我们在 Chroot 环境里面创建了一个 UID 为 500、名为 alice 的用户。在本地系统上，有一个 UID 为 500、名为 se 的用户。虽然这两个用户的 UID 都是 500，但是它们之间没有任何关系。alice 的信息是在 Chroot 环境里面的，在外边是体现不出来的。所以在本地系统上查看 alice 的文件，由于 UID 的缘故，它的 Owner 变成了 se。关于这个问题我们在前面已经解释过了。

```
[root@redhat03 ~]# chroot /export/chroot/
bash-4.1# id alice
uid=500(alice) gid=500(alice) groups=500(alice)
bash-4.1# ls -l /home/alice/.ssh/authorized_keys
-r----- 1 alice alice 815 Jun 30 10:00 /home/alice/.ssh/authorized_keys

[root@redhat03 ~]# id se
```

```
uid=500(se) gid=500(se) groups=500(se)
```

```
[root@redhat03 ~]# ll /export/chroot/home/alice/.ssh/authorized_keys  
-r----- 1 se se 815 Jun 30 10:00 /export/chroot/home/alice/.ssh/authorized_keys
```

```
[root@redhat03 ~]# ll -n /export/chroot/home/alice/.ssh/authorized_keys  
-r----- 1 500 500 815 Jun 30 10:00 /export/chroot/home/alice/.ssh/authorized_keys
```

然后，我们在容灾端 redhat02 上创建了一个 UID 为 501、名为 se 的用户。执行 rsync 同步后就出现了有趣的事情。

```
[root@redhat02 ~]# id se  
uid=501(se) gid=501(se) groups=501(se)
```

```
[root@redhat02 ~]# ll /export/chroot/home/alice/.ssh/authorized_keys  
-r----- 1 se se 815 Jun 30 10:00 /export/chroot/home/alice/.ssh/authorized_keys
```

```
[root@redhat02 ~]# ll -n /export/chroot/home/alice/.ssh/authorized_keys  
-r----- 1 501 501 815 Jun 30 10:00 /export/chroot/home/alice/.ssh/authorized_keys
```

rsync 在同步时无法识别 Chroot 环境。对于 rsync 来说，它们只是一堆需要复制的文件而已。每一个文件都带有属性。由于 authorized\_keys 的 Owner 在本地读取时是 500，而系统外面恰好有一个 UID 等于 500 的用户 se，同步过程中 Owner 就被 rsync 给标记成 se 并带过去了。而容灾端上也有一个 se，不过它的 UID 是 501，到了目的地后 authorized\_key 的 Owner 就变成 501 了。这和 Chroot 环境里面的 passwd 的描述是不一致的，从而导致了登录失败。

从这个案例中可以看到，如果 SFTP Server 上没有 se 账户，或者两台服务器上 se 用户的 UID 与 GID 都是一样的，就不会引起歧义了。如果两边拥有同名用户，且它们的 UID 和 GID 不一致，就会因信息不符而导致登录失败。

此外还有 Chroot 环境差异的影响。如果两边环境都是用本章所提供的方法创建的，就不会有问题。如果生产端是自己创建的，而容灾端的创建又使用了本章所提供的方法，则有可能产生环境差异。你可以尝试着用生产端的 sshd\_config 配置文件替换掉容灾端上的同名文件来解决。

## 11.4 本章小结

本章为大家总结了文件共享服务在不同应用场景下的使用，以及如何构建 WebDAV、NFS 和 SFTP 服务。下一章，我们将进入一个新的篇章——系统运维篇。届时，我将和大家一同聊聊日常运维工作中的那些事儿。



## 第 12 章

# 硬件故障告警与维修

“不积跬步无以至千里，不积小流无以成江海。”这是荀子在《劝学》中的名句，我们上学时都背通过。老人家教育我们，做事都是靠一点一滴的积累才能有所成就。不单成就要靠积累，反过来灾难也是一样的。硬件故障的告警与维修本是运维工作中最不起眼的一件小事，不过当生产环境的服务器数量达到一定规模后，也许你就会和下面这位老黄一样，有着相同的烦恼。

### 运维故事 13：老黄的烦恼

七月的盛夏，空气中仿佛燃起了熊熊烈焰，如同炼钢炉一般炙烤得令人窒息。在一家名为星宇的咖啡厅内，店员们都有显得有些百无聊赖。在这样的烈日炎炎之下，并没有太多的顾客上门。唯有最远端的一个角落里，坐着两个男人，正在低声地交谈着。

“老黄，我可得恭喜你啊。听说你最近调到信息中心当大官儿去了。”

“哎，你可别拿我取笑了。什么大官儿，不过是一个小班子罢了。”

“嘿嘿，这次岗位调整，银子没少涨吧？”

“咳，别提了。天底下哪有那么好的事儿啊，这银子可不是白拿的，我都快愁死了。”

“哦？”对方显然是对此深感惊讶，马上把身子探了出去。“怎么个情况？你倒是说来听听。”

“兄弟，你不知道，我这些天的日子不好过啊。”被对方称作老黄的那个中年人，喝了一口咖啡后，继续诉苦。

“原来领导跟我讲，带技术团队希望让我多历练历练，谁想到这是个烫手的山芋。这个团队是以前老严玩剩下的，内部问题比较多，人心涣散。这不，上个月我刚转岗过来，屁股还没坐稳，就出了一堆糟心的事。”

对方一听就明白了。老黄以前是做业务出身的，不懂技术。前些年他转了管理岗，本来干得风生水起。可现如今，领导让他接替老严下面的一个技术团队，这倒是大姑娘上轿——头一回。“不过，老黄你带团队不是还挺有一套的嘛，慢慢来，别着急。”





“我还没说完呢。我原来在业务干了这么多年，也总结了一套自己的管理之道。原以为来这儿喝喝茶聊聊天，把人际关系和团队气氛搞好也就得了，谁承想根本就行不通。管人我是绝对有把握，但是解决技术问题我是一窍不通。技术这东西不像人，丁是丁，卯是卯，差一点儿也过不去啊。”

“我们部门是活儿多人少，平时工作采取轮班制，一人值一周的班。你拿设备维修来说就是个苦差事。上次机房巡检发现了一大批故障机，我的手下老张是当周的值班员，他连带分析问题，和各个部门确认维修时间，再加上汇总整理，前后花费了三天的时间，总算是赶在周末前发送了报修邮件。周一早晨一上班，厂商那边就派单了。正轮到小夏值班，小夏是一问三不知，也不清楚哪些设备该修，哪些不该修，结果弄了半天也搞不清楚。最后没办法，又全都推回到老张这边了。打那儿以后，老张也学乖了。因为谁报修的设备，最后就得谁去处理。既然如此，谁还那么积极？大家都是能拖就拖。结果今天不巧有一台设备因为没有及时处理，导致了线上故障。我也为此挨了批评。”

## 12.1 硬件故障的特点

如果我们从长期的观察角度来看，会发现硬件故障有两大特点。

第一，部件损坏的范围比较集中。硬件设备的部件虽然很多，但硬件故障却只集中在几个点上。最容易出问题的是机械硬盘和阵列卡，其次是内存、电源和 SSD。

第二，故障发生的时间点相对集中。这是为什么呢？因为你一开始采购硬件设备时，厂商就是按照批次生产的。如果在某一个批次的产品上发现质量缺陷，那么肯定就不止一两台有问题。硬件设备整机质保一般为三年，当临近这个时间点的时候，该批次的产品就会接二连三地出毛病。此外，检修不及时的问题在一些企业里也是存在的。平日不烧香，临时抱佛脚。比如说下周领导要来视察了，大家赶紧提前做个巡检吧。结果，这不检查还不要紧，一检查发现了一大堆故障告警。

硬件故障分为停机维修和非停机维修两种。除了更换硬盘和电源模块外，其他故障都需要停机操作。如果是停机维护，就要进行业务切换，进而影响到 SLA 指标。当然，非停机维修也不一定就对业务一点儿影响都没有。例如，数据库对于 I/O 性能很敏感，即便是更换硬盘，也要考虑在业务低谷期实施。

要降低硬件故障对 SLA 的整体影响需从两个方面入手：第一把好质量关，选择优质的设备；第二，硬件故障必须及时通告，不可全仗着人工巡检，更不能等着从业务故障上去反推。有些业务挂了，上去一检查才发现原来是硬件故障。

大多数不可逆的硬件故障在发生前都有明显的征兆和提示（比如阵列卡电池的问题、磁盘坏块、内存读写错误等），而且像磁盘、电源这些部件都是冗余的，所以由硬件故障所导致的停机，通常都是因为前面积累了太多的问题却没有及时处理而造成的。类似于这种因

为一块坏盘没及时维修而导致线上故障，让原本一个非停机维修事件变成了生产事故，实在是太不应该了。

## 12.2 硬件故障告警

硬件故障告警有多种形式，但归根结底都要走带外管理这条通道。通过操作系统去监控硬件故障存在很多弊端，首先要安装大量的硬件管理工具，然后通过不断轮询才能实现硬件故障检测。这样做不但占用业务资源，效果也未见得好，其全面性、即时性以及准确性都与带外管理的主动告警有着不同程度上的偏差，更何况创建管理这些监控项也很花精力。

本节将为大家讲述带外管理的几种告警形式的应用场景，以及事件类型与告警级别的选择。

### 12.2.1 告警方式

服务器和存储的带外管理为用户提供了多种告警方式，常见的包括 E-mail、SNMP 和 Syslog。下面我们就来了解一下它们的使用场景。

#### 1. E-mail

E-mail 的告警方式比较适合于少量节点的服务器，它没有什么部署成本，开销也很低。服务器的硬件故障率低于 3%，所以使用 E-mail 告警不必担心扩展性和可靠性，即便支撑几万个节点也完全没有问题。但我为什么说它不适合大规模场景呢？虽然 E-mail 故障告警很准，但是它的信息量太少，只能给出 SN、带外管理 IP 和基本的硬件故障信息。而且邮件不太适合与其他系统联动，除了告警以外没有其他辅助支持。就像我们在第 6 章中所评论的低效告警一样，只知道狼来了，但其他周边情况就不清楚了。另外，E-mail 的时效性不高。虽然收发邮件很快，但通常邮件检查的间隔时间都不会太频繁。在硬件故障很多的情况下，这种方式的工作效率会比较低。

E-mail 告警在少于 3000 节点时可以作为主要告警形式存在，在生产环境的建设初期，由于监控平台的不完善以及人员匮乏，它也是一种快速的替代方案。当生产规模超过 3000 节点时，它应该只作辅助告警。

发送 E-mail 告警时，发件人的 IP 地址是带外管理地址，邮箱账户却是固定的，很多邮件安全策略不允许发送这样的邮件。可以考虑在两者之间增加一台邮件网关作为转发。如若希望提升可靠性并降低运维成本，可以采购硬件邮件网关设备。邮件网关至少应当有两台，它们的前端应该部署负载均衡设备，将邮件网关的地址指向一个虚拟地址。如果指向的是一个域名，则需要为带外管理配置一个可用的 DNS Server。





## 2. SNMP 告警

SNMP Trap 是传统的监控告警，适合融入监控平台。很多互联网公司的监控平台都是自主开发的，最好能够与 CMDB 实现联动，这样在告警时能携带更多的相关信息。我们在第 6 章中曾经举过这样的例子。一台主机发生硬件故障后，简单的一个告警通知只能告诉我们哪里发生了什么故障，但却无法得知所属 Owner、业务关系、告警等级和故障风险。告警发生后，因为信息缺失的缘故，运维团队成员的工作状态依然停留在查询和沟通上。我们的告警平台如能和 CMDB 实现联动，则可以在最短时间内开始故障处理。

## 3. Syslog

Syslog 将故障以日志形式远程发送到日志服务器上，同时引入 ELK Stack 对日志进行实时查询和分析。Syslog 不是单纯的告警形式，它的主要目的是对多条信息进行综合分析和结果收敛。这种结果产出会有一定的延迟，但它是值得的。如果监控中心在很短的一段时间内，收到了多条来自于不同主机的故障告警，你会如何去看待这样的问题？也许你可以很快处理完所有的故障，但却无法立即回答如下这些问题。为什么会有这么多故障？它们有什么共同点？它们是来自同一业务、同一品牌还是同一机柜？发生这些故障的真正原因是什么？进一步的解决方案又是什么？

一说到硬件故障，大家会觉得不就是硬件坏了需要更换吗？如果单独去看每一条故障信息，这种想法没有错。但实际上，你在对大量故障告警进行综合分析后才会发现，有些黑锅让硬件来背实在是非常冤枉。我自己就遇到过很多类似的情况。比如，由于在 BIOS 里启用了 C State，很多设备在计算需求高峰时出现了 CPU 错误。还有，因为功耗限电设置导致大量服务器的磁盘出现坏块。在发布结论之前，类似于这样的关系，我们不可能自己一条条地去梳理，需要日志分析系统帮助我们做出正确的判断。

### 12.2.2 事件类型和告警级别

不同厂商对事件类型和级别的定义并不完全相同。在此之前，请你先将相关项调研清楚。常见的事件类型包括：传感器运行状态、存储类、配置类、登录审计类或者其他。常见的事件级别包括：Informational、Warning、Error、Critical、Emergency/Fatal。

不必将所有内容都配置到告警范围内，否则会导致告警信息泛洪。告警过度，人一旦处理不过来，可能也就不会再去看了。我们的值班手机上曾创造过 1300 多条未读短信告警的记录，长假期间也曾收到过 9 万余封的告警邮件，这些都是告警信息泛洪的典型事例。削减不必要的监控项以及告警收敛是减少泛洪的两个有效手段。

类型只要选择传感器运行状态和存储类事件即可，请将 Error 或者 Critical 作为直接触发告警的级别。对于未经判断的 Warning 事件，不要直接纳入告警范围之内。这个原因稍后我们会详细分析。



## 12.3 硬件故障分析

当 SE 遇到硬件故障告警或者服务器宕机时，首先要完成信息采集工作，然后对其进行故障分析，以判断问题的根源在哪里。本节将为大家介绍一些常见的故障分析手段与故障分析案例。

### 12.3.1 常用分析手段

硬件故障的类型不同，其采集方式也不同。信息源通常来自于传感器状态、日志、终端输出等几个部分，采集它们需要会用到 IPMI、MCE 和阵列卡管理工具。

#### 1. IPMI

IPMI 是采集硬件故障最直接的一种手段，它可以获取传感器状态和 SEL 日志。调用 IPMI 接口的工具叫作 `ipmitool`，它支持远程和本地两种查询方式。如果你希望使用本地查询，请确保操作系统已经执行过如下命令。最好在自动化部署系统时，就将它们写入 `/etc/rc.local` 文件，确保每次开机后都能生效。

```
# modprobe -a ipmi_devintf
# modprobe -a ipmi_si
# /etc/init.d/ipmievtd start
```

使用 `ipmitool` 检查 SEL 日志时，请先用 `elist` 列出所有的信息。

```
[root@station103 ~]# ipmitool sel elist
1 | 09/09/2014 | 08:49:51 | Event Logging Disabled SEL | Log area reset/cleared |
  Asserted
2 | 07/13/2015 | 16:34:42 | Power Supply PS Redundancy | Redundancy Lost
3 | 07/13/2015 | 16:34:44 | Power Supply Status | Power Supply AC lost | Asserted
```

命令执行后，得到的是所有的日志。分析日志时，可根据时间戳来定位，然后用 `get` 命令去查询某一条日志的详细信息。例如，我想分析 2015 年 7 月 13 日的某一条日志，可以执行如下这条命令。

```
[root@station103 ~]# ipmitool sel get 2
SEL Record ID       : 0002
Record Type         : 02
Timestamp           : 07/13/2015 16:34:42
Generator ID        : 0020
EvM Revision        : 04
Sensor Type         : Power Supply
Sensor Number       : 74
Event Type          : Generic Discrete
Event Direction     : Assertion Event
Event Data (RAW)    : 01ffff
Description         : Redundancy Lost
```

```

Sensor ID           : PS Redundancy (0x74)
Entity ID           : 7.1
Sensor Type (Discrete): Power Supply
Unable to read sensor : Device Not Present

```

```

FRU Device Description : OEM fru (ID 17)
Unknown FRU header version 0x00

```

如果你遇到了和电压、电流、温度有关的故障，可以使用 `sdr` 命令来检查各个传感器的运行情况。检查一下风扇有没有坏的，电源是不是掉电了，等等。还可以使用 `senor get` 命令来查询某一个传感器的详细信息。

```

# ipmitool sdr
# ipmitool sensor list
# ipmitool sensor get <SENSOR_NAME>

```

```

[root@station103 ~]# ipmitool sdr |grep 'Fan1A RPM'
Fan1A RPM           | 3360 RPM           | ok

```

```

[root@station103 ~]# ipmitool sensor list |grep 'Fan1A RPM'
RPM           | 3360.000           | RPM           | ok           | na           | 720.000           | 840.000           |
na           | na           | na

```

```

[root@station103 ~]# ipmitool sensor get 'Fan1A RPM'

```

Locating sensor record...

```

Sensor ID           : Fan1A RPM (0x30)
Entity ID           : 7.1
Sensor Type (Analog) : Fan
Sensor Reading       : 3360 (+/- 120) RPM
Status               : ok
Lower Non-Recoverable : na
Lower Critical        : 720.000
Lower Non-Critical    : 840.000
Upper Non-Critical    : na
Upper Critical        : na
Upper Non-Recoverable : na
Assertion Events      :
Assertions Enabled    : lnc- lcr-
Deassertions Enabled  : lnc- lcr-

```

## 2. MCE

MCE (Machine Check Exceptions) 是一个用来检查并记录 CPU、内存以及总线错误的工具。如下几种情况都会触发 MCE 事件。

- ☐ 系统总线错误;
- ☐ 处理器缓存错误;
- ☐ 内存寻址错误;
- ☐ ECC 校验问题;

### □ 温度过载。

默认情况下，MCE 日志会记录在 /var/log/mcelog 中。触发 MCE 后，日志中就会写入类似下面的内容。

```
Hardware event. This is not a software error.
MCE 0
CPU 0 BANK 1
MISC 35082189 ADDR 1b5829a001
TIME 1429259817 Fri Apr 17 16:36:57 2015
MCG status:
MCi status:
Corrected error
Error enabled
MCA: MEMORY CONTROLLER GEN_CHANNEL unspecified_ERR
Transaction: Generic undefined request
STATUS 900000400012008f MCGSTATUS 0
MCGCAP 1000c18 APICID 40 SOCKETID 1
CPUTID Vendor Intel Family 6 Model 47
```

MCE 事件分为两种。一种是警告级事件，当它发生时写入 MCE 日志。还有一种是致命级事件，当它发生时系统来不及记录就已经挂掉了。此时，MCE 消息会输出到 TTY 上。这时，请先用手机拍照存留。待系统复原后，将上述内容输入一个文本中（例如名为 ERROR\_FILE），执行如下命令进行现场复原。

```
# mcelog --ascii < ERROR_MESSAGES_FILE
```

### 3. MegaCli

存储类故障是最常发生的，分析处理它们离不开阵列卡管理工具。LSI 使用的是 MegaCli 和 StorCli。HP 的阵列卡管理工具使用的是 hpacucli 和 hpssacli。通常处理存储故障的方法是：先检查磁盘和阵列卡的当前状态，然后收集并分析相关的日志信息。

我们以最常用的 MegaCli 来举例，使用 -ShowSummary 可以直观地反映出控制器、电池、PD 以及 VD 的当前状态。

```
[root@station103 ~]# MegaCli64 -ShowSummary -aALL |egrep -i "Controller|BBU|Enclosure|Status|PD|Connector|Virtual|^ *State"
Controller
    Status                : Optimal
BBU
    BBU Type              : BBU
    Status                : Healthy
Enclosure
    Status                : OK
PD
    Connector             : 00<Internal><Encl Pos 1 >: Slot 0
    State                 : Online
    Connector             : 00<Internal><Encl Pos 1 >: Slot 1
    State                 : Online
```



```

Connector      : 00<Internal><Encl Pos 1 >: Slot 2
State          : Online
Connector      : 00<Internal><Encl Pos 1 >: Slot 3
State          : Online
Connector      : 00<Internal><Encl Pos 1 >: Slot 4
State          : Online
Connector      : 00<Internal><Encl Pos 1 >: Slot 5
State          : Online
Connector      : 00<Internal><Encl Pos 1 >: Slot 6
State          : Online
Connector      : 00<Internal><Encl Pos 1 >: Slot 7
State          : Online
Virtual Drives
Virtual drive   : Target Id 0 ,VD name
State          : Optimal

```

PD 预定义的状态有如下几种。

- ☐ Unconfigured Good;
- ☐ Unconfigured Bad;
- ☐ Online;
- ☐ Offline;
- ☐ Foreign;
- ☐ Failed;
- ☐ Rebuild;
- ☐ Hot Spare。

Unconfigured 是磁盘位于 VD 之外的一种常见状态, 它代表硬盘和控制器是连通的, 但尚未加入任何一个 VD 内。例如, 刚刚插入空闲槽位的新盘都是这种状态。

当一块磁盘被创建成 VD 的一部分后, 已经激活了的就是 Online, 尚未激活的则是 Offline。如果这块磁盘以前在其他控制器上做过配置, 再加入现有 VD 成员时, 就会被标记成 Foreign。例如你用一块旧盘替换 VD 中的一块坏盘的时候。严格地说, Foreign 并不是状态, 它代表磁盘上有元数据, 但元数据不属于本地, 而是来自于其他控制器。

VD 预定义的状态有如下四种情况。

- ☐ Optimal;
- ☐ Partially Degraded;
- ☐ Degraded;
- ☐ Offline。

Optimal 是指 VD 内所有的 PD 成员都处于 Online 状态。Offline 则代表由于 VD 内的 PD 成员发生故障, 导致 VD 故障数据无法访问。Partially Degraded 和 Degraded 指降级故障, 表明 VD 内的 PD 成员发生故障, 但 VD 尚未损坏。两者的区别是: 前者在当前状态下,

还可以继续承受一个或者多个 PD 成员的损坏。你可以把它们想象成 RAID 6 和 RAID 5 在损坏一块磁盘时 VD 分别所处的状态。

-ShowSummary 只是一种快速简单的查询手段，如果你想进一步分析，需要更多的命令支持。以下这些命令可用于对阵列卡和磁盘进行全面检查，包括状态查询和日志收集。

```
MegaCli64 -adpCount
MegaCli64 -AdpAllInfo -aALL
MegaCli64 -FwTermLog -Dsply -aALL
MegaCli64 -PDList -aALL
MegaCli64 -PDGetNum -aALL
MegaCli64 -EncInfo -aALL
MegaCli64 -LDInfo -Lall -aALL
MegaCli64 -LdPdInfo -aALL
MegaCli64 -LDGetNum -aALL
MegaCli64 -CfgDsply -aALL
MegaCli64 -CfgFreeSpaceinfo -aALL
MegaCli64 -AdpEventLog -GetEventLogInfo -aALL
MegaCli64 -AdpEventLog -GetEvents -f events.log -aALL
MegaCli64 -AdpEventLog -GetSinceShutdown -f shutdown.log -aALL
MegaCli64 -AdpEventLog -GetSinceReboot -f reboot.log -aALL
MegaCli64 -AdpEventLog -IncludeDeleted -f deleted.log -aALL
```

### 12.3.2 常见故障错误分析

本节我们将列出一些常见硬件故障错误并加以分析，和读者一同讨论遇到这些错误时该如何看待和处理。

#### 1. 电池电量告警

前面我们提到未经判断的 Warning 事件不要直接纳入告警范围之中，因为原生的 Warning 事件很多情况下并非一定是硬件故障。下面就是一个典型的 Warning 事件。从表面上看，它表达的是控制器电池的电量不足了。但我们没有必要过度紧张，它和你的手机电池不一样，这是采用锂电池方案充放电时的一个正常现象。

【Warning】The PERC1 battery is low.

在讨论这个问题之前，我们先来了解一下电池为什么要进行充放电操作。阵列卡的电池解决方案有锂电池和镍氢电池两种。电池储能是一个化学反应的过程，不管是哪一种方案，电池在自然状态下都会缓慢地自放电，电量在一段时间后就会有所下降。

但两种电池方案的特性有所不同。锂电池的惰性较强，为了能够及时校准电量，避免电池因为自放电而导致电量不明确，阵列卡控制器要对锂电池进行主动的、周期性的充放电操作，以激活内部的化学物质，从而保证电量的准确性，同时还可以判断电池是否发生故障或者老化。它的充放电过程是这样的：首先将电池内的电量完全放光，然后再进行充电操作，直至电量充满，最后完成电量的重新计算校准。由于电池在放电结束时电量为 0，

无法为控制器缓存供电，控制器会将缓存关闭，从而导致读写性能急剧下降。控制器的这种行为是受到了默认策略 No Write Cache if bad BBU 的影响。而镍氢电池的方案则不太一样，它并不需要通过完全放电来校准电量。当控制器检测出电量降低到某一阈值时就开始充电了。整个充电的过程中无须关闭缓存，所以并不会影响 I/O 性能。

很多新一代的服务器都不再使用锂电池的方案了。当这个事件发生时，它只是提醒用户，如果数据中心此时突然掉电，电池中的电量无法为阵列卡缓存支撑足够的时间。这一点完全不必担心，现代数据中心采用的都是双路供电，加之柴油发电机在 30s 之内就可以启动，基本上已经不再是什么问题了。

可以使用 Megacli 命令检查阵列卡的缓存策略。如果你想在任何情况下，都强制开启控制器缓存，需将策略调整成 Write Cache OK if bad BBU。

```
[root@station103 ~]# MegaCli64 -LDGetProp -Cache -LALL -aALL
Adapter 0-VD 0(target id: 0): Cache Policy:WriteBack, ReadAhead, Direct, Write
Cache OK if bad BBU

Exit Code: 0x00
```

如果遇到电池电量持续告低，则有可能是电池损坏的前兆。阵列卡的电池寿命大多在两年以上，比服务器的生命周期略短一些。关于这个 Warning 事件，我个人认为可采取延后观察的策略。如果这个警告在几个小时之内还没有消除，再将其通告出来。如果该告警每隔几天就触发一次，需手动执行充放电操作后观察，短期内再次发现告警，也需要及时处理。这两种情况出现时都要更换电池，请尽快完成业务切换。

## 2. 链路告警

类似于下面这种链路问题，多是由人为操作或网线质量引发的，而网卡故障率很低，极少可能是一个硬件问题。适合的做法是将链路监控的工作交给网络去处理。网络监控关注的是链路状态，如果链路恢复后也可以及时知晓。

```
[Warning] The NIC Integrated 1 Port 4 network link is down.
[Info] The NIC Integrated 1 Port 4 network link is up.
```

## 3. 预判告警

你可能在很多服务器上都能看到类似如下的这种情况。故障预判技术是为了保证磁盘在故障发生之前就警示用户，让用户有充足的应对时间。它针对磁盘内部构件的一些属性进行测量，以此来预测故障。

```
[Warning] Predictive failure reported for Disk 1 in Backplane 1 of Integrated
RAID Controller 1.
```

当这个告警消息出现时，可以使用如下命令来检查磁盘预判故障的错误数，以决定是否更换新的磁盘。



```
MegaCli64 -PDList -aAll|grep "Predictive Failure Count"
```

不过故障预判对基于 RAID 的盘阵来说意义不大，不管你是提前换还是坏了以后换，更换硬盘给系统带来的影响是一样的。唯一的区别是：提前换的准备时间更充裕，你可以选择在业务低谷期执行这项任务。而那些已经完成了业务拆分的生产环境则完全不必有这样的担心。

首先，盘阵在 Rebuild 时读写性能会有 15%~25% 的下降。业务完成拆分后，单节点的日常压力很低，基本上不会受到 Rebuild 的影响。其次，假设你的盘阵采用了 RAID 10，在第一块盘出现故障后，依旧有 2/3 的概率允许损坏第二块盘。现在很多厂商都能做到 4 小时之内的本地紧急响应服务，并能提供充足的备件，坏盘后再更换都是来得及的。再次，预判故障也需要一定的数量积累，并不能因为一两个错误，就认为磁盘一定是有问题的。

我们在这里关于 Warning 事件做一个小结。此类告警的出现频率比较高，在遇到它们时请不要过度解读。当它们出现时，只是告诉你可能存在故障，但并非一定就是故障。这时需要配合更多的条件进一步加以分析，分析时尽可能借助工具或脚本进行辅助判断，当达到触发条件后再向用户发送告警。这一点对于 E-mail 告警尤为重要。如果你的服务器数量较少，且短期内没有其他告警手段，可以考虑开启告警模式。假如情况是相反的，请注意一定要关闭告警模式。

#### 4. 电力告警

如果大部分服务器告警，请检查数据中心的电力情况。如果是单台服务器告警，大多与电源线或者电源模块有关。出现最后一个错误时，可以通过升级固件尝试解决。

```
[Critical] The power input for power supply 2 is lost.  
[Critical] The storage BP1 Power cable is not connected, or is improperly connected.  
[Critical] The system board PS1 PG Fail voltage is outside of range.
```

#### 5. 检查 CPU 或内存的错误

下面这些是 CPU 或者内存方面的错误，通常它们的出现都会伴随着 MCE 事件的触发。

```
[Critical] CPU 1 machine check error detected.  
[Critical] CPU 2 has an internal error (IERR).  
[Critical] Correctable memory error rate exceeded for DIMM_A2.  
[Critical] Multi-bit memory errors detected on a memory device at location(s) DIMM_B3.
```

#### 6. Kernel Panic

Kernel Panic 是一种严重的故障。它的发生代表操作系统在遇到内部致命错误时，无法安全处理而采取的一组动作。引发 Kernel Panic 的常见原因有两种。一种来自于文件系统的故障，关键字是 File system、mount、VFS 等。另一种来自于硬件错误，大多和 CPU、内存以及寻址有关。但也不排除是应用程序造成的问题。一般出现 Kernel Panic 时，系统会停止响应，然后自动重启或者等待用户手动干预。系统会向磁盘提供一份内核转储文件 vmcore 用于事后分析。下面简单介绍 vmcore 的基本分析方法。

启用转储功能，需要事先安装软件包 `kexec-tools`，并启用 `kdump` 服务。当触发 Kernel Panic 时，默认会将转储文件 `vmcore` 存放到 `/var/crash/` 目录下。`vmcore` 的大小和系统环境有关，如果 `/var/crash/` 目录不是单独挂载的，建议将它调整到一个独立分区当中，以免转储文件过大导致根目录写满。如下所示，我们将转储文件的数据路径改成了 `/export/` 目录。

```
[root@station103 ~]# grep ^path /etc/kdump.conf
path /export
```

我们通过如下命令可以模拟出一个 Kernel Panic，并从控制台观察消息输出，如图 12-1 所示。

```
echo c > /proc/sysrq-trigger
```

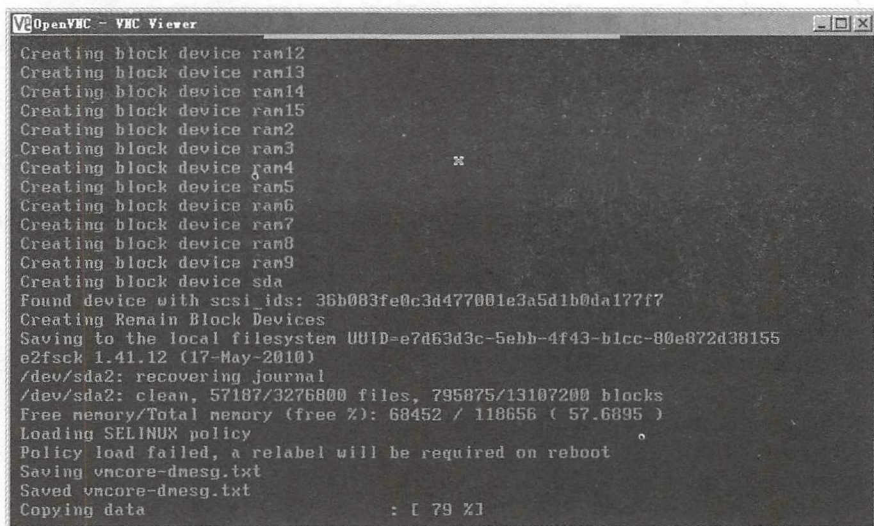


图 12-1 Kernel Panic

待系统重启后，会在 `/export/` 目录下产生一个名为 `vmcore` 的转储文件。它保存了系统崩溃时的内核地址、进程状态等内容。

```
[root@station103 ~]# ll -lh /export/127.0.0.1-2017-07-18-16\16:55/vmcore
-rw----- 1 root root 132M Jul 18 16:17 /export/127.0.0.1-2017-07-18-16:16:55/vmcore
```

分析 `vmcore` 要使用到 `crash` 命令，`crash` 用来读取转储文件并进行现场复原。如果你的系统没有这个命令，请安装软件包 `crash`。执行 `crash` 之前还要安装对应内核版本的软件包 `kernel-debuginfo-common` 和 `kernel-debuginfo`。这两个软件包可以在 <http://debuginfo.centos.org> 下载。

以下是我为大家准备的一个 `crash` 故障分析的例子。假设某台服务器出现了 `crash` 的故障，重启后，我们对生成的 `vmcore` 文件进行故障分析，找出导致 `crash` 的根本原因是什么。首先，执行命令 `crash` 载入 `vmcore` 文件，做故障场景的状态复现。

```
[root@station103 ~]# crash /usr/lib/debug/lib/modules/2.6.32-431.el6.x86_64/
vmlinux /export/127.0.0.1-2017-07-18-16:16:55/vmcore
```

```
crash 6.1.0-5.el6
Copyright (C) 2002-2012 Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010 IBM Corporation
Copyright (C) 1999-2006 Hewlett-Packard Co
Copyright (C) 2005, 2006, 2011, 2012 Fujitsu Limited
Copyright (C) 2006, 2007 VA Linux Systems Japan K.K.
Copyright (C) 2005, 2011 NEC Corporation
Copyright (C) 1999, 2002, 2007 Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002 Mission Critical Linux, Inc.
This program is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it under
certain conditions. Enter "help copying" to see the conditions.
This program has absolutely no warranty. Enter "help warranty" for details.
```

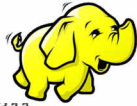
```
GNU gdb (GDB) 7.3.1
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu"...
```

```
KERNEL: /usr/lib/debug/lib/modules/2.6.32-431.el6.x86_64/vmlinux
DUMPFILE: /var/crash/127.0.0.1-2017-07-18-16:16:55/vmcore [PARTIAL DUMP]
CPUS: 32
DATE: Tue Jul 18 16:16:51 2017
UPTIME: 00:05:42
LOAD AVERAGE: 0.19, 0.12, 0.05
TASKS: 642
NODENAME: station103.example.com
RELEASE: 2.6.32-431.el6.x86_64
VERSION: #1 SMP Fri Nov 22 03:15:09 UTC 2013
MACHINE: x86_64 (2600 Mhz)
MEMORY: 64 GB
PANIC: "Oops: 0002 [#1] SMP " (check log for details)
PID: 3171
COMMAND: "bash"
TASK: ffff880823195540 [THREAD_INFO: ffff880822fb4000]
CPU: 17
STATE: TASK_RUNNING (PANIC)
```

在提示符下执行 `bt` 命令找到系统崩溃的地址段，搜索关键字 `exception`。根据关键字找到 #8 并记录下关键信息 `exception RIP: sysrq_handle_crash+22`。

```
crash> bt
PID: 3171 TASK: ffff880823195540 CPU: 17 COMMAND: "bash"
#0 [ffff880822fb59e0] machine_kexec at ffffffff81038f3b
#1 [ffff880822fb5a40] crash_kexec at ffffffff810c5d92
```





```
#2 [ffff880822fb5b10] oops_end at ffffffff8152b510
#3 [ffff880822fb5b40] no_context at ffffffff8104a00b
#4 [ffff880822fb5b90] __bad_area_nosemaphore at ffffffff8104a295
#5 [ffff880822fb5be0] bad_area at ffffffff8104a3be
#6 [ffff880822fb5c10] __do_page_fault at ffffffff8104ab6f
#7 [ffff880822fb5d30] do_page_fault at ffffffff8152d45e
#8 [ffff880822fb5d60] page_fault at ffffffff8152a815
[exception RIP: sysrq_handle_crash+22]
RIP: ffffffff8134b6c6 RSP: ffff880822fb5e18 RFLAGS: 00010096
RAX: 0000000000000010 RBX: 0000000000000063 RCX: 000000000000fe7f
RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000063
RBP: ffff880822fb5e18 R8: 0000000000000000 R9: ffffffff81645da0
R10: 0000000000000001 R11: 0000000000000000 R12: 0000000000000000
R13: ffffffff81b01a40 R14: 0000000000000286 R15: 0000000000000004
ORIG_RAX: ffffffff81b01a40 CS: 0010 SS: 0018
#9 [ffff880822fb5e20] __handle_sysrq at ffffffff8134b982
#10 [ffff880822fb5e70] write_sysrq_trigger at ffffffff8134ba3e
#11 [ffff880822fb5ea0] proc_reg_write at ffffffff811f328e
#12 [ffff880822fb5ef0] vfs_write at ffffffff81188f78
#13 [ffff880822fb5f30] sys_write at ffffffff81189871
#14 [ffff880822fb5f80] system_call_fastpath at ffffffff8100b072
```

然后执行 `dis` 命令查询出错代码的位置。查询时可以使用错误信息或者寄存器地址。

```
// 根据错误信息查询
crash> dis -l sysrq_handle_crash+22
/usr/src/debug/kernel-2.6.32-431.el6/linux-2.6.32-431.el6.x86_64/drivers/char/
sysrq.c: 130
0xffffffff8134b6c6 <sysrq_handle_crash+22>:      movb    $0x1,0x0
// 也可以根据寄存器地址查询,这和上面的结果是一致的。
crash> dis -l ffffffff8134b6c6
/usr/src/debug/kernel-2.6.32-431.el6/linux-2.6.32-431.el6.x86_64/drivers/char/
sysrq.c: 130
0xffffffff8134b6c6 <sysrq_handle_crash+22>:      movb    $0x1,0x0
```

通过调试我们发现,原来引发 `crash` 故障和源代码 `sysrq.c` 中的 130 行的上下文有关。据此我们得知,要去 `sysrq.c` 中检查相关的源代码,进一步分析故障原因。

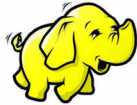
## 7. 电池故障

遇到下面几种故障,有可能你的 BBU 需要进行更换。针对第二条故障,请先检查 BBU 的状态,尝试手工充放电。

```
[Critical] The PERC1 battery has failed.
[Critical] The battery on Integrated RAID Controller 1 can no longer recharge.
[Critical] Integrated RAID Controller 1 is unable to recover cached data from
the Battery Backup Unit (BBU).
```

## 8. 磁盘巡检故障及坏块

如果遇到如下这些错误,则代表你的磁盘发生了故障或者存在坏块。如果多块盘都出现了类似的错误,很可能是连线或者背板的问题,如果是单盘则多为磁盘本身的问题,需



要更换。

```
【Critical】Fault detected on drive 6 in disk drive bay 1.  
【Critical】Patrol Read found an uncorrectable media error on Disk 0 in Backplane  
1 of Integrated RAID Controller 1.  
【Critical】An unrecoverable disk media error occurred on Disk 0 in Backplane 1  
of Integrated RAID Controller 1.  
【Critical】Bad block medium error is detected at block 0x9d9daf06 on Virtual  
Disk 2 on Integrated RAID Controller 1.  
【Critical】Virtual Disk 0 on Integrated RAID Controller 1 has failed.
```

这里要说明的一点是，Patrol Read 是一种提前检测硬盘驱动器错误的方式，能够在驱动器故障威胁到数据完整性之前就发现错误。

### 9. 修复过程中出现的告警错误

以下这些告警是在更换坏盘的过程中产生的，请检查磁盘与背板之间的连线是否紧固，如果物理连接没有问题，则可确认该磁盘或其接口部分发生了损坏。

```
【Critical】Drive 10 is removed from disk drive bay 1.  
【Critical】The rebuild of Disk 5 in Backplane 1 of Integrated RAID Controller 1  
failed due to errors on the source physical disk.  
【Critical】The rebuild failed due to errors on the target Disk 4 in Backplane 1  
of Integrated RAID Controller 1.
```

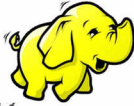
## 12.4 传统维修的问题

传统故障报修的难点就在一个多字。故障设备多、填报内容多、干系人多、处理情况多、流程多、品牌多、数据中心多。

大多数产品的设备年硬件故障率维持在 2%~3%，较少一部分则可以达到千分之几，这样的数据看上去还蛮不错的。也就是说，假设你有 10 000 台服务器，按照 3% 的年故障率计算，一年的设备维修量也就只有 300 次而已。但是，你如果只是简单地做加减乘除，认为这些故障平均每天发生一次，那就大错特错了。我们已经讲过了，硬件故障的特点是按波次出现的。平日里不来是不来，一来就是一堆。我处理故障最多的一次是一天 25 个。在这个过程中不但要收集大量信息并加以分析，还要填写除故障详情以外的很多相关信息，包括联系人、数据中心地址、设备 SN 等。你可以想象一下，一天处理几十个故障是一种怎样的体验。然而这还不算什么，因为你的噩梦才刚刚开始。

SE 不负责业务，到底是下线维修还是延迟处理不是他能决定的。所以，SE 在发送报修邮件之前，必须先与 Owner 确认是否可以下线关机。这么多的设备对应的业务和 Owner 自然也各不相同。其确认结果往往是：一些设备可以维修，而另一些则不能。你就要把那些暂时无法维修的设备挑出来并做好标记。即便所有的故障设备都可以维修，它们也可能来自不同的品牌或者不同的数据中心，报修时依旧需要将它们拆分开来。除此之外，你还会





遇到这样一种情况：Owner 允许故障设备报修，但由于他们不具备应用切换条件（比如没有备机、没有业务拆分等），只能在一个很短且特定的时间窗口内进行维修。故障设备不能提前关闭，SE 要等到厂商的工程师到达现场后，才能打电话通知 Owner 关机。当故障设备关闭时，SE 还得负责确认并通知厂商进行维修。这样的工作效率实在是太差了。

除了这种拉锯式的过度沟通，信息不畅也是传统维修的一大问题。就像本章开篇中那个故事所述的一样：老张报修了一批设备，等到厂商派单时，却是同事小夏接听的报修电话。但小夏并不清楚相关情况，如果他要负责处理，就必须先弄清楚这件事是谁发起的。如果设备没有全部修完，等到下次接电话时可能又变成了其他人。事情处理到什么阶段了？当前是什么情况？就更没有人清楚了。

我们看，大量时间被浪费在信息分拣、手工录入和低效沟通上了。处理一台故障设备尚且如此波折，那么一大波故障设备向你靠近时，其工作压力让人几近崩溃。

## 12.5 报修系统的需求定义

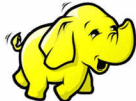
看起来，现在是要构建一个报修系统来解决这些棘手问题的时候了。报修系统最终要解决三大难题——信息分拣、辅助录入和事件通告。

既然所有的数据都存储在 CMDB 中，信息分拣的工作就不应该再依靠手工处理，报修系统要具备信息查询、分拣以及归类的自动化功能。人员在数据录入时只需输入必要信息，其余内容可从 CMDB 的查询结果中返回并自动填入。流程的各项环节在完成后，应该给干系人发送通告，并注意输出数据的收敛和重排。

当然，解决问题也不能完全依靠报修系统，流程化简也是一项同等重要的工作。就拿限定维护时间窗口这个事情来说，没有备用节点以及业务缺乏有效拆分使得故障节点不能及时下线，导致厂商和 Owner 这两个原本不相干的角色在维修过程中却建立了荒唐的紧耦合关系。设备维修要等着 Owner 关机下线后才能操作，修一台就得等半天，而且还连累了 SE 和 IDC 的人。因为 Owner 并不和厂商直接对接，如果你的 IDC 还是外包服务，那 SE 就要全程协调并跟踪处理，SE 在业务、IDC 和厂商之间成为了一个传话筒。维修一个设备要牵扯到四波人，何况这样的设备还有一大批？你要是故事中的老张，肯定也吃不消啊。

所以，最好的解决办法就是任务拆分，谁的机器谁来修。当然，这个方案的实现前提是业务完全分布于多个节点之上，不必担心某一个节点的故障影响过多的流量。当一个故障告警发送出来，SE 首先对其进行分析处理。如果确认需要维修，SE 会完成一系列的信息收集、故障填报等工作。接下来，SE 会将任务发布给 Owner，让 Owner 根据实际情况评估后，酌情做出处理。原则上，故障节点应当及时下线。如有特殊情况也可延缓维修计划，待条件具备后再关闭故障节点。我们不要错误认为，设备只要没坏就不能停。因为你现在不停，早晚也得停。主动的人为干预远比将来发生故障所带来的影响要小得多。我想这样





一个道理大家都应该明白。

SE 在将任务发布给 Owner 前要做好前置工作。

第一，要给 Owner 关机的权限。Owner 完成业务下线后，关机不过是举手之劳的事情。如果再返回来找 SE 关机，这不是多此一举吗？

第二，SE 要把所有报修信息都填写好，只要 Owner 允许故障节点维修，就自动给厂商发送报修邮件。自始至终，整个维修事件的流程应当是一步一步朝前走的，不要在任何环节上出现拉锯扯皮的现象。如图 12-2 所示，当流程完成简化后，无谓的交互环节将大大减少。这个报修流程中牵扯到了 NOC。NOC (Network Operations Center) 是一个集管理、维护、监控及协调于一身的平台中心，而本章中的 NOC 特指狭义概念上的监控平台。因为关闭故障设备前要告知 NOC 同时关闭对应的告警项，所以报修系统会与 NOC 之间有一些互动。图 12-3 所展示的是故障设备在维修流程的各个环节中所处的状态，以及需要通告的干系人有哪些。

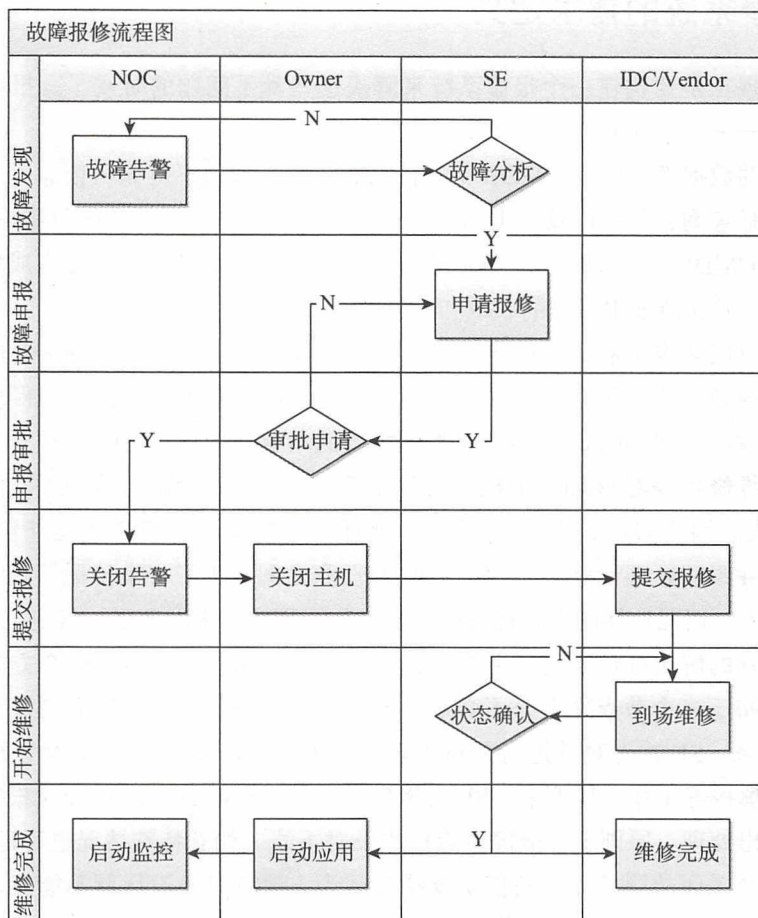
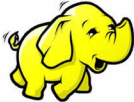


图 12-2 故障报修流程图



|      |   | 待审批  | 已延迟  | 待维修  | 已完成   |
|------|---|--|--|--|---|
| 故障申报 | <br>SE |  |  |  |   |
| 申报审批 |   | <br>Owner | <br>Owner |  |   |
| 提交报修 |   |  |  | <br>Everyone      |   |
| 开始维修 |   |  |  | <br>SE IDC/Vendor |   |
| 维修完成 |   |  |  |  | <br>Everyone |

图 12-3 流程状态及干系人关系图

接下来，我们针对报修系统的需求定义做进一步的详细讨论。

## 12.5.1 故障申报环节的设计需求

故障申报是报修流程中的第一个环节，也是最为重要的一步操作。这里需要解决信息填报和任务分层的问题。

### 1. 信息填报

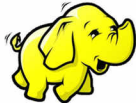
信息填报有两个具体的需求。

第一，提供足够的相关信息。信息不全是导致无效沟通的主要因素，很多无谓的沟通都是因为任务委派时没有交代清楚造成的。你报修一台设备，没有提供 SN，就得不到技术支持；没有提供地址和联系人，人家就不知道怎么去，到了地方不知道要找谁。最后出了问题还是要打电话来问，这些都是无谓的沟通。

申报信息应当包括业务 IP、带外管理 IP、SN、设备品牌及型号、故障类型、设备位置（数据中心地址、机柜号）、干系人及其联系方式、日期时间、详细故障信息（错误提示、日志、截图等）。

第二，要尽量减少人工填写的内容。如果信息填报全靠人工来完成，这样的报修系统不用也罢。申报人应当可以在业务 IP、带外管理 IP 和 SN 三者之间填写任意一项，然后由系统自动补全其他已知信息。除了故障类型和详细故障信息需要手工填写以外，其他内容的输入都应该能与上述三者产生联动。因为这些内容本来就是存储在 CMDB 当中的。

为了防止信息缺失或错误导致的异常，自动填写的区域应当允许手工修改，需要手动填写的区域应当检查空值。详细故障信息的填写，应该支持文本输入和附件上传两种方式。



对于附件上传，我们建议增加一个功能：允许报修系统对附件名称增加 SN 作为前缀。这个功能应作为一个可选项，让用户根据实际情况选择开启或者关闭。

## 2. Case 和 Task

在第 6 章我们已经讲过 Case 和 Task 的区别，以及设置它们的重要性。SE 在发起故障申报的时候，应当允许他们在同一个申报工单里填写多个报修设备。每一个设备对应的是一个 Task，不同的 Task 会有不同的干系人，而工单则作为一个 Case 来发布。当申报人填写完第一个 Task 后，可以选择继续添加下一个 Task，待所有 Task 全都填写完毕后，再统一提交 Case 即可。

这些 Task 可能需要拆分处理。比如设备的下线时间不同、其对应的 Owner 和所属品牌也有所不同。就像你网购一样，一个订单就是一个 Case，订单里面可以包含多个来自不同商家的商品，有些是自营的，有些则是第三方的，系统对于这些不同的 Task 要进行拆分。同样的，Case 由 SE 创建，每增加一个报修设备就自动创建一个 Task。干系人只需关心自己的 Task 就好，Task 完成后会由 SE 确认关闭。而 Case 则是在所有的 Task 全都完成后自动关闭的，无须人为干预。如果没有任务分层的设计，每出现一个故障，就要填一个工单，那岂不是要把人活活累死？

### 12.5.2 审批通告环节的设计需求

故障申报实现了合并操作，而邮件通知的发送则需要拆分。就好像你买了一部自营的手机，又买了很多第三方的图书，不同的商品要发送给不同的商家来处理。干系人不应在收到邮件通知后，还得在 Case 里去找属于自己的那个 Task。你见过哪个商家在收到订单时还能看到用户在别家购买的商品？我们前面已经强调过了，干系人并不关心其他人的 Task，他只要处理好自己的那一部分就行了。

在审批环节中，主要干系人（也就是审批人）是故障设备的 Owner。Owner 可根据实际情况同意维修或者延迟维修。这与一般的审批概念不同，不存在拒绝维修这种选项。如果审批人选择了延迟维修，此时故障设备的状态将变更为“已延迟”，但依旧要保持在审批环节当中。对于同一类型申报项的审批，应当允许审批人实现批量操作的功能，以此来提升工作效率。

### 12.5.3 提交报修环节的设计需求

当审批人同意维修某一个 Task 后，报修系统应当自动发送报修邮件，并完成对内部相关干系人的通告工作。注意：审批人每完成一个审批行为后，就会触发邮件发送和通告行为。报修系统应当引导审批人，让他尽可能批量通过所有允许维修的 Task。当发送报修邮件时，报修系统依旧要考虑信息处理的问题。



### 1. 邮件合并

例如，三个审批者分别提报了五台待维修的服务器，详情如表 12-1 所示。报修系统应当根据故障设备所属品牌进行分类。同属供应商 A 的服务器 Server11 和 Server12 应当进行邮件合并处理，而服务器 Server21、Server22 和 Server23 则有所不同。虽然它们同属供应商 B，但供应商 B 会收到两封报修邮件。这是因为它们所属的 Owner 不同。两个 Owner 自然就会发出两封邮件。对于供应商 B 来说就是两个 Case。

表 12-1 维修设备列表

| Server   | SN     | IDC    | Vendor | Owner |
|----------|--------|--------|--------|-------|
| Server11 | ASN001 | 朝阳数据中心 | A      | Alice |
| Server12 | ASN002 | 朝阳数据中心 | A      | Alice |
| Server21 | BSN001 | 朝阳数据中心 | B      | Bob   |
| Server22 | BSN002 | 大兴数据中心 | B      | Bob   |
| Server23 | BSN003 | 大兴数据中心 | B      | Eric  |

需要注意的一点是，在邮件合并后，附件也会被合并到一起。如果申报人在上传附件时没有对文件名进行标识，邮件合并后就无法区分哪个附件是属于哪个服务器的。这就是我们要求报修系统给附件名称增加 SN 前缀的原因。

### 2. 信息分层及过滤

批量报修时的 Task 会很多，有必要对信息进行分层显示。顶层的 Task 只展示最基本的必要信息即可。如表 12-2 所示，我们给出设备型号、SN、机架位置、带外管理 IP 和故障类型即可。详细信息可以放在后续部分或者附件当中。

表 12-2 故障服务器顶层汇总信息

| Machine Type | SN     | Location | OOB IP  | Fault Type    |
|--------------|--------|----------|---------|---------------|
| AN380        | ASN001 | Rack201  | 1.1.1.1 | Power failure |
| AN300        | ASN002 | Rack513  | 1.1.1.2 | Disk failure  |

由于 CMDB 里关联了很多内部的敏感信息，例如业务类型、业务 IP 地址以及 Owner 的联系方式等，因此在发送邮件时要做到内外有别，不能将这些内容暴露出去。一方面是出于安全考虑，另一方面这些内容对维修也没有任何帮助，反而容易干扰邮件的阅读。

### 3. 信息拆分

多个 Task 在进行邮件合并时，位于不同数据中心的故障设备应当被拆分。例如，Bob 审批通过了 Server21 和 Server22 的维修，报修邮件的格式应当是如下这个样子的。

- Hello 及通用寒暄用语；
- 位于朝阳数据中心的地址及其联系人（见表 12-3）；

- ❑ Server21 的故障信息汇总及详情；
- ❑ 位于大兴数据中心的地址及其联系人（见表 12-4）；
- ❑ Server22 的故障信息汇总及详情。

表 12-3 朝阳数据中心地址及联系人

| IDC    | Address | Contact | Mobile | E-mail            |
|--------|---------|---------|--------|-------------------|
| 朝阳数据中心 | xxx     | Grace   | 138000 | grace@example.com |

表 12-4 大兴数据中心地址及联系人

| IDC    | Address | Contact | Mobile | E-mail            |
|--------|---------|---------|--------|-------------------|
| 大兴数据中心 | yyy     | Robot   | 153000 | robot@example.com |

### 12.5.4 设备维修环节的设计需求

由于前置工作都已完成，因此在维修环节中没有过多需求。维修系统要做的只有一件事：当 SE 确认维修完毕并将状态修改成“已完成”后，发送内部通告。

### 12.5.5 数据查询统计的设计需求

任何一套系统都离不开查询和统计的功能。查询通常发生在流程执行期间，而统计则是被用于事后总结。

#### 1. 信息分级

因为报修系统涉及的信息非常多，信息分级依旧是首要需求。顶级信息将以汇总的形式来显示，应当只包含最基本的内容。例如，业务 IP、带外管理 IP、SN、设备品牌及型号、故障类型、设备位置和流程状态。其他细节则以子项的形式收敛在下面。例如，流程信息（干系人、各环节的触发日期）、详细故障信息等。子项的展示形式有浮动窗口、弹出式窗口和折叠式几种。浮动窗口无法复制信息内容，而且它要求光标必须停靠在特定的位置上，也就是所谓的热区。如果信息项排列得比较密集就很容易出错，你以为自己指向的是 item A，而实际展示的却是 item B。弹出式窗口和主页面是分离的，如果要查询的子项太多，相互切换时容易发生混淆。我建议采用最后一种形式，需要时可通过点击加号图标展开子项信息，不需要的时候可以将其收起来。

#### 2. 丰富的查询条件、排序和统计功能

不同角色对于报修系统的查询需求是不同的。例如，业务方和运维团队最关心设备的维修情况；架构师和测试团队则更关注硬件故障的分布规律；而管理者需要通过维修系统了解下属的工作情况。

所以，我们不能仅从设备维修这样一个维度上去考虑问题。报修系统应当支持多个维度的数据查询条件，包括按照业务 IP、带外管理 IP、SN、操作人、日期、流程状态等条件

来查询。查询结果应当支持排序,并且要能实现对机房、设备品牌及型号、故障类型、操作人、日期、流程状态的数据统计。

## 12.6 本章小结

本章我们关于硬件故障告警与维修这个话题,总结了故障告警的两大特点,论述了硬件故障的分析手段,还分享了很多实际案例。后半部分,我们分析了硬件维修工作的难点,就如何构建一套报修系统,详细解析了其中的业务需求。通过一系列的详尽讨论,我们看出故障告警的主要特点是量大,而故障处理的重点、难点在于告警的收敛、业务的信息拆分以及流程的优化。



## 第 13 章

# 主机系统信息安全基础

做金融业务离不开各种支付牌照，这些牌照是公司的命脉，有了它们你才能获得相关金融业务的经营许可。所以，配合安全部门完成证照的安审工作也是 SE 的一项重要任务。

信息安全对我来说并不陌生，我入行就是从安全做起的。这一晃，已经过去了整整十年的光景。信息安全是一个很广泛的话题，本章我们将围绕操作系统层面，讨论一些信息安全的基础知识。

### 13.1 系统安全加固的基本要求

本节将为读者介绍系统安全加固的一些基本要求，这些内容主要来源于各项安全法案的标准条例。不管是信息安全等级保护、PCI (Payment Card Industry) 还是 SOX (Sarbanes-Oxley Act)，这些基本要求都是大同小异的。

#### 1. 最小化系统

最小化系统是安全配置的首要任务。例如，按需安装软件包，禁用不必要或有风险的服务、端口和账户等。这些都是老生常谈，不必过多赘述。在这里，我只想提一下第 7 章中有关镜像部署系统的问题。如果在制作镜像时为了偷懒，只编译一个 All in one 的 image 是不妥的。因为这个镜像文件涵盖了全部场景所需的软件包，完全没有场景区分。而有些漏洞不一定非得要求服务是运行着的，你只要安装了某些包或者库文件，威胁就会存在。安装它们也许只在某种特定场景下才会发生。如果你没有做场景区分，则所有的主机都会受到影响。这样，维护的工作量就会大幅增加，同时也违背了最小化系统的原则。

#### 2. 账户管理

系统登录者的身份必须是可识别的，但如果系统账户采取实名制管理，维护起来就很成问题。大企业人员流动频繁，要在所有生产系统上不断地增删用户太不现实了。系统账户应当基于用户角色来创建，不同的用户角色去对应不同的权限，这一点可通过 sudo、chown 和 ACL 来实现。

在办公区客户端和生产服务器之间，需要一个堡垒机设备（Jump Server）。客户端不允许直接访问生产系统，必须先登录堡垒机，然后通过堡垒机跳转到生产系统上。堡垒机的账户应当是实名制的，也就是按照员工 ID 建立账户。堡垒机本来就是做安全防护和审计的，而且为数不多，在它上面维护实名账户，不论是管理、同步还是审计都相当方便。堡垒机的实名账户和系统账户之间是相互映射的。登录后的账户、操作和时间戳都有记录，满足了身份识别和审计的要求。

### 3. 密码策略

#### (1) 定义密码生命周期的更新策略

其配置文件为 `/etc/login.defs`。主要参数及其定义如下所示。

```
PASS_MAX_DAYS // 密码的最大生命周期（超过这个期限就要更换密码）
PASS_MIN_DAYS // 密码的最小生命周期（当前密码最少要使用多久）
PASS_MIN_LEN  // 最小密码长度
PASS_WARN_AGE // 密码过期前几天发出告警（告警信息会在登录时提示）
```

#### (2) 定义基于 PAM 认证的密码修改策略

其配置文件为 `/etc/pam.d/system-auth`。PAM 的语法格式如下所示。

```
service type control module-path module-arguments
```

`service` 指代的是 `pam.d` 目录下的文件名，它可以为空，但必须是小写。`type` 包括了 `account`、`auth`、`password` 和 `session`。`control` 用于控制 PAM-API 的最终行为，常用的有如下这几种。

```
required // 认证必须成功才可以，但认证失败后不会立即通知应用程序，而要等到 stack（service
           和 type）中的所有模块全部执行完毕后才可
requisite // 和 required 类似，不同的是，当它认证失败后会立即通知应用程序
sufficient // 认证成功后不再进行其他认证，反之尝试其他模块的认证方法
optional  // 认证成功与否都不影响
```

简而言之，`required` 和 `requisite` 是模块认证的必要条件，必须通过才可以。`sufficient` 属于充分不必要条件，它与其他模块认证之间是逻辑或的关系。`optional` 的认证结果则是无关紧要的。

PAM 语法中的最后两部分是具体的模块和参数。我们可以使用下面这条命令来查找和密码策略相关的模块及用法。

```
[root@station101 ~]# man -k pam |grep password
pam_cracklib      (8) - PAM module to check the password against dictionary words
pam_pwhistory     (8) - PAM module to remember last passwords
pam_unix          (8) - Module for traditional password authentication
password-auth-ac [system-auth-ac] (5) - Common configuration files for PAMified
services written by authconfig (8)
```

其中，`pam_cracklib` 是最常用的定义密码策略的模块，其主要参数包括如下这些内容。

```

minlen           // 最小密码长度
difok            // 历史密码个数
retry           // 重试次数
minclass         // 最少密码类型组合数
ucredit          // 大写字符数
lcredit          // 小写字符数
dcredit          // 数字字符数
ocredit          // 其他字符数
maxrepeat        // 同一字符的最多重复次数
maxclassrepeat   // 同一字符类型的最多重复次数
maxsequence      // 连续字符的最多次数
enforce_for_root // 强制 root 用户也遵守密码策略

```

ucredit、lcredit、dcredit、ocredit 的取值范围既可以是正值，也可以为负值。正值代表该类字符最多可以包含多少个，负值代表该类字符最少必须要有多少个。

maxrepeat、maxclassrepeat 和 maxsequence 用来控制重复字符、字符类型和连续字符的最大值，用于防止用户设置类似 aaa 或 123 这样的弱口令。

root 用户默认是不受密码策略约束的。enforce\_for\_root 可强制让 root 也遵守相应的规则。

### (3) 定义加密算法

Linux 系统的密码是以加密形式存储在 shadow 文件中的。人们已经证明 MD5 不再安全，应当使用 SHA-256 或者更强的哈希算法。关于哈希算法的相关介绍以及策略定义的具体操作，我们将在 13.4.1 节中作详细说明。

## 4. 防病毒

虽然 Linux 平台很少会感染病毒，但依旧不能排除这种可能性。ClamAV 是一个开源、轻型的反病毒解决方案，它是由 clamd、clamav 和 clamav-db 这几个软件包组成的。ClamAV 有两个非常重要的配置文件：/etc/clamd.conf 用于定义扫描选项，/etc/freshclam.conf 用于定义病毒库的更新策略。文件中的配置项非常多，我们只选取部分重要的参数来介绍。

clamd.conf 的主要参数如下所示。

```

OnAccessExcludePath // 除了该目录以外的所有子目录都会被扫描
ExcludePath          // 禁止扫描的目录，尽可能不去扫描纯数据文件或 jar 包，那会非常耗时
MaxConnectionQueueLength // 限制最大连接队列，数字过大会影响线上性能
MaxThreads           // 限制最大线程数，该数值设定对性能有影响
ScanArchive          // 依据实际情况可以不去扫描归档文件，以减少扫描时间

```

freshclam.conf 的主要参数如下所示。

```

DatabaseMirror // 下载病毒库的服务器地址
Checks         // 每天检查病毒库版本的次数

```

其他配置项的定义请参考 ClamAV 的用户手册，手册的获取可从如下链接下载。

<https://github.com/vrtadmin/clamav-faq/raw/master/manual/clamdoc.pdf>



ClamAV 通过 clamd 服务于后台运行工作。clamd 服务启动后，首先要检查病毒库版本，并尝试将其升级到最新。手工升级可通过执行 freshclam 命令来完成，病毒库的版本检查可通过执行 clamd --version 命令来完成。

不论是从安全还是从性能的角度上考虑，不可能所有客户端都去连接外部网络完成病毒库的升级。因此要在本地构建一个病毒库服务器，具体的实现方法可以参考如下链接。

<https://github.com/Cisco-Talos/clamav-faq/blob/master/mirrors/CvdPrivateMirror.md>

## 5. 入侵检测

为防止程序或文件的非授权变更，线上系统应当部署变更检测机制，并且每周至少巡查一次。AIDE 全称是 Advanced Intrusion Detection Environment，即高级入侵检测环境。它的工作原理很简单，通过初始化数据库保存检测对象的 Metadata，然后用它定期和检测对象最新的 Metadata 作比较，以此判断是否发生过非法改动。

AIDE 的配置文件 /etc/aide.conf 很容易理解，它主要定义了两部分内容——属性和检测对象。属性包括对象的权限、inode、链接数、用户、组、时间戳、哈希值等。多个属性可以构成一个 Group。Group 的作用就是为一组属性设置一个别名，并支持嵌套，Group 的名称注意要大写。定义检测对象时要在后面跟上属性或者 Group。你可以使用如下命令获取 AIDE 的示例用法。

```
# egrep -v "^#|^$" /etc/aide.conf
```

AIDE 并不适合监控那些容易发生变动的内容。例如，etc 目录下的配置文件就是典型的实例。建议你只针对系统命令和标准库进行检查，因为它们是最容易被替换成后门程序的。另外，如果有自己开发的程序或者服务，也应当列入监控范围，以防被人插入恶意代码。完整性检查不应过度，除了性能会受到影响外，信息泛洪也不利于问题的处理。

在完成配置文件的定义后，还需要进行数据库的初始化，然后安排定期的变更检查即可。具体操作方法如下所示。

```
// 初始化数据库
# aide -i
# mv /var/lib/aide/aide.db.new.gz /var/lib/aide/aide.db.gz
// 检查文件变更
# aide -C
// 检查并更新数据库
# aide -u
```

如果在检测过程发现问题，应当立即下线问题节点，并重装一台新的主机。当然，侵入原因必须调查清楚，以作为修复问题的依据，但问题节点不能回到线上。Linux 系统一切皆文件，恶意代码可以放到任何地方。当你发现系统某处有问题时，很可能还有别的地方也被改掉了，重置是最好的解决方式，而不是尝试着去修复它。

## 6. 配置文件标准化

刚刚我们提到 AIDE 不适合监控配置文件的变更，是因为配置文件变动的可能性很大，而 AIDE 数据库的更新比较慢，所以不推荐采取检测机制来确保配置文件的标准化。

使配置文件标准化的最佳方式是使用 Puppet。节点会定期向 Puppet Master 发送请求，并完成状态同步工作。有关配置文件的变更，只要在 Puppet Master 上执行就可以了。如果节点上的配置文件被非法篡改，也会在同步周期内自动修正回来。

## 7. 审计日志

审计日志包括用户登录、执行操作和时间戳等内容。按照合规要求，本地日志应当集中转储到具有备份条件的存储上。这些可能会被误解为是“马后炮”的行为。事实上并非如此，审计工作并不总是滞后的。下面将为大家介绍将历史命令记录到日志文件的方法。如果将其转发到日志服务器并进行实时分析，即可实现针对特定危险行为的预警效果。

首先，在全局配置文件 `/etc/profile` 的末尾添加如下内容。

```
export PROMPT_COMMAND='{ msg=$(history 1 | { read x y; echo $y; }); logger -p
user.info "##" "$msg" "##" $(who am i);}'
```

然后编辑配置文件 `/etc/rsyslog.conf`，按照如下方式将 `user.info` 日志从 `messages` 重定向到 `cmd.log`。

```
[root@station101 ~]# egrep "^*.info|^user.info" /etc/rsyslog.conf
*.info;mail.none;authpriv.none;cron.none;user.none /var/log/messages
user.info /var/log/cmd.log
```

完成后重启 `rsyslog` 服务并重新创建新会话，新会话中的所有操作都将被记录到 `cmd.log` 中。可使用 `tailf` 命令跟踪 `cmd.log`，以检测审计功能是否生效，日志内容和格式如下所示。注意：日志中有两个时间戳，第一个是日志的写入时间，第二个是命令操作的时间。

```
Aug 25 15:35:50 station101 root: ## date ## root pts/0 2017-08-25 15:29 (10.1.2.3)
Aug 25 15:35:51 station101 root: ## pwd ## root pts/0 2017-08-25 15:29 (10.1.2.3)
```

## 13.2 关于安全配置的反思

我曾经在实施项目中遭遇过这样一件事：我们部署了一套应用系统，按照甲方安全部门的要求，系统上线前要先经过他们的安全测评。在测评过程中，安全部门使用了一款知名的扫描器设备对我们的系统进行了检查。我在他们之后给出的反馈报告里，竟然看到了这样一条安全建议——将某配置文件权限改为 700。我不知道这个建议的依据是什么。抛开这个设置是否影响服务正常运行不说，单就执行权限而言，它对于一个配置文件的意义又在哪里呢？

时代在不断发展，一些安全配置的科学性和可行性确实值得商榷。但因为各种历史原



因，它们依旧存在于各种标准当中。本节中，我会就其中部分内容和大家共同探讨。这些内容纯属学术性质的，与行业制度和法律法规无关，还请各位不要对号入座。

### 13.2.1 慎用账户锁定

对于大规模生产环境来说，一定要慎用账户锁定。账户锁定策略是针对穷尽攻击的一种消极防御，使用不当反而会给自己带来无尽的麻烦。穷举攻击是针对已知账户名展开的，例如禁止使用 root 登录 SSH 服务就是这个道理。但是普通账户完全没有锁定的必要。如果不是从内部泄密，账户信息相对于攻击者来说是未知的，在这种情况下盲目展开穷举攻击是毫无意义的。当然，这并不是重点。即便攻击者从某种途径掌握了一些相关信息，那么此时的账户锁定策略反而成了最大的帮凶。如果我用错误的密码不断地发起登录尝试，系统账户岂不是会被一直锁定么？最终结果就是谁也登录不了。使这个手段可以让 Owner 暂时失去远程控制权，再配合上一些破坏性操作的话，我想后面的事情就用不着详细描述了。

面对穷举攻击，最有效的做法是通过网络设备阻塞攻击源，而不是采取锁定账户这种被动的、“含羞草式”的防御策略。通过搜索安全日志 /var/log/secure 可以找出登录失败者的 IP 地址，应根据事件的触发次数定义阻塞攻击者 IP 的阈值。防御的总体原则是增加攻击者的成本。毕竟，不断地更换 IP 地址是不容易的。如下所示就是 secure 日志中关于登录失败的信息。

```
Sep 7 13:55:52 station101 sshd[25371]: Failed password for root from 10.1.2.3
port 36029 ssh2
Sep 7 13:55:06 station101 sshd[25371]: Failed password for root from 10.1.2.3
port 36029 ssh2
Sep 7 13:55:09 station101 sshd[25371]: Failed password for root from 10.1.2.3
port 36029 ssh2
Sep 7 13:55:09 station101 sshd[25372]: Connection closed by 10.1.2.3
```

### 13.2.2 密码的烦恼

操作系统发展了这么多年，密码依旧是最基本的验证方式，在这一点上没有发生过任何改变。密码的重要性自不必说，但是它的安全性一直是困扰人们的一个难题。首先，密码很容易被摄像头、Sniffer、键盘木马等方式截获。其次，多套密码难于管理。还有，提升设定密码的难度会迫使人们使用更加不安全的应对措施。

#### 1. 苛刻的复杂度检查

为了防止人们使用弱口令，复杂性定义有时显得过于苛刻，甚至可以用变态来形容。好像不管你怎么设置，都难以同时满足复杂又好记的需求，而且很多系统都需要密码验证。有些人只用一套密码来登录所有的系统，于是埋下了撞库的隐患。有些人由于记不住那些复杂性密码，就将它们存储到各种不安全的介质之上，反倒成了更加危险的事情。这里我要提一下那些所谓的密码存储软件，很难说它们完全没有风险。就我个人而言，一般的密



码可以用软件保存，但最好不要写完整，删掉其中一段。密码录入时手动补全剩余的部分，以防密码泄露。至于核心密码，绝对要记在心里。

## 2. 烦人的更新策略

除了复杂度以外，频繁的更新策略也令人厌烦。事实上，这些策略检查毫无意义，完全是形同虚设，稍微变通一下就能绕过去。比方说，你要求 90 天修改一次密码，同时设置了历史密码为 5。那好，我事先准备 5 个临时密码，然后连续修改 6 遍，最后还不是重新用回了原来的密码么？看似非常合规，实际上自欺欺人而已。

## 3. 设置一个好密码

还是那句话，安全意识大于一切策略。既然更新检查的意义不大，重点还是要回到密码的设计上。一个好密码真的就只能是一组难以记忆的字符串吗？其实不然，好密码也能记住，但要注意几点原则，以免落入弱口令的陷阱之中。

首先，丢弃那些尽人皆知的“弱智”变换与组合。不要小看你的对手，他们的想象力比你丰富得多。比如，123@qwe、1qaz@WSX、P@ssw0rd 这类密码，它们都是复合要求的，但你能说这样的密码是安全的么？依此类推，就不要再再用类似的方式去设计你的密码了。只要模式类似，字典库在拓展后很容易就能覆盖掉你自认为完美的字符串组合。

其次，任何密码都带有其使用者的“原始含意”。用户在设计密码时会带入自己熟悉的内容，这是因为熟悉的东西才好记，但这非常危险。有些密码看似严谨，而熟悉使用者的人却很容易破解。假设你是某个明星的狂热粉丝，了解内情的攻击者就会尝试着将相关信息纳入猜想队列中。

只要不按常理出牌，就没有规律可循。计算机的逻辑性很强，但它无法处理未知异常。如果你的逻辑过于奇葩，它就完全拿你没办法。弱口令扫描是基于字典或事先定义好的逻辑来展开攻击的。如果抓不住规律，我相信没有人愿意花很多精力去对抗一个随机事件。短时间内无法得手的话，入侵者会选择知难而退。因为不断地耗下去对他来说没有任何好处，反而会增加自己暴露的几率。

## 4. 双因子验证

对于一定要使用密码的场景，最好能配合动态口令，以防止别人窥视或者嗅探你的密码。这也是很多法案强制的要求。

目前大多数令牌（Token）只能显示动态口令，登录时还是要人工输入，显得非常麻烦。不过也有基于 USB 接口的 uToken，它可以通过触发按钮来完成口令的自动填写。希望将来的 Token 能像数字证书一样，登录时自动请求动态口令，岂不善哉？

## 13.2.3 sudo 的意义

关于 sudo 的使用，我们总能听到这样一种说法：使用 root 账户管理系统是不安全的，平时应以普通用户身份操作，需要执行特权命令时，再使用 sudo 来提权。如果你是这个系



统的管理者，这句话根本就没有道理。允许 root 登录 SSH 确实存在着账户被穷举的风险，可使用 root 账户管理系统的风险又在哪里呢？

SE 有很多管理工作都会涉及特权命令，如果都用 sudo，那得添加多少个命令？如果把所有的特权命令都加进去，那和不用 sudo 有什么区别？如果账户存在弱口令，一个添加了所有特权命令的 sudo 账户就比 root 安全么？如果担心误操作，多敲一个 sudo 就能避免了么？

sudo 的作用是委派特权命令给普通用户的，是当普通用户有特殊需要，我们又不能向其交付 root 账户时，所使用的一种解决方案。它不是什么保险，和系统安全也没多大关系。sudo 列表是一个白名单，应当只添加少数命令才对。有些系统管理者，给 sudo 一个 ALL Command 的权限，美其名曰是为了安全操作，甚至干脆支持 sudo 到 root，这不是自欺欺人吗？

## 13.3 sudo over LDAP 的实现

sudo 是管理员将特权命令委派给普通用户的一种解决方案。sudo 本地的配置文件是 /etc/sudoers，但本地模式不适用于管理大量的节点。sudo 的配置是动态的，账户角色会根据需要随时增减特权命令。当节点数量超过 1000 台时，使用配置管理工具同步会产生很大的时间差，无法确保 sudo 的及时生效。基于 LDAP 集中式验证的解决方案非常适合处理这个问题。

### 13.3.1 服务端配置

下面以 station103.example.com 为例，为大家介绍如何构建一台用于 sudo 的 LDAP 服务器。

#### 1. 服务端安装

首先在 station103 上安装如下软件包，并复制配置文件的模板。

```
# yum install -y openldap openldap-servers openldap-clients
# cp /usr/share/openldap-servers/slapd.conf.obsolete /etc/openldap/slapd.conf
# cp /usr/share/openldap-servers/DB_CONFIG.example /var/lib/ldap/DB_CONFIG
# cp /usr/share/doc/sudo-1.8.6p3/schema/OpenLDAP \
/etc/openldap/schema/sudo.schema
```

#### 2. 设置 LDAP 的管理密码

接下来，我们要设置 LDAP 的管理密码。当我们执行修改查询 LDAP 信息的操作时，需要提供管理密码来验证 LDAP 管理员的身份。这里我们使用的密码是 redhat。在执行完 slapasswd 命令后，会生成一个哈希值，请一定要将它保存好。在接下来的步骤里，会将这个哈希值添加到 LDAP 的配置文件中。

```
[root@station103 ~]# slappasswd
New password:
Re-enter new password:
{SSHA}ZN8ZlUbdz5kVnSLQxTeE2Rb7d14R9Q3w
```

### 3. 修改配置文件

编辑 LDAP 配置文件 `/etc/openldap/slapd.conf` 中的如下几行内容。其中 `suffix` 可以看作一个顶级域，也就是这个 LDAP 架构的根节点。`rootdn` 是管理根，可以看作 LDAP 的 `root` 账户，`rootpw` 就是设置 LDAP 管理密码时最后生成的哈希值。配置管理 LDAP 时，需要提供 `rootdn` 和管理密码进行验证。

```
[root@station103 ~]# egrep '^root|^suffix|^log|sudo' /etc/openldap/slapd.conf
include      /etc/openldap/schema/sudo.schema // 引用 sudo.schema 文件
suffix       "dc=example,dc=com"
rootdn       "cn=Manager,dc=example,dc=com"
rootpw       {SSHA}ZN8ZlUbdz5kVnSLQxTeE2Rb7d14R9Q3w
loglevel 1   // 这一行增加在 argsfile 下面
```

在 `rsyslogd` 配置文件 `/etc/rsyslog.conf` 中，增加如下命令定义日志，并重启 `rsyslog` 服务。

```
[root@station103 ~]# grep local4 /etc/rsyslog.conf
local4.* /var/log/ldap.log
```

### 4. 测试配置文件及服务是否生效

如果以下命令都能执行成功，且 TCP 389 端口监听打开，则表示 LDAP 服务运行正常。

```
# service rsyslog restart
# service slapd start
# chkconfig slapd on
# rm -rf /etc/openldap/slapd.d/*
# slaptest -f /etc/openldap/slapd.conf -F /etc/openldap/slapd.d
# chown -R ldap:ldap /etc/openldap/slapd.d/
# service slapd restart
```

// 上述命令执行完毕后，再检查 TCP 389 是否已经开启监听。

```
[root@station103 ~]# netstat -tlnp |grep slapd
tcp      0      0 0.0.0.0:389          0.0.0.0:*           LISTEN   22894/slapd
tcp      0      0 :::389              :::*                 LISTEN   22894/slapd
```

### 5. 编辑并导入自定义的 `sudo.ldif` 文件

我们在这里创建了一个名为 `sudo.ldif` 的 `ldap` 文件，意为为普通用户 `user` 定义一个 `grep` 的特权。`user` 原本就可以执行 `grep` 命令，但前提必须是 `user` 有读取权限的文件，像密码文件 `shadow` 就不可以。如果给 `user` 授予了 `sudo`，它就可以使用 `grep` 来搜索 `shadow` 文件中的内容了。

以下是 `sudo.ldif` 的具体配置。

```
[root@station103 ~]# cat sudo.ldif
```





```

dn: dc=example,dc=com
objectClass: top
objectClass: domain

dn: ou=sudoers,dc=example,dc=com
objectClass: top
objectClass: organizationalUnit
ou: sudoers

dn: cn=user,ou=sudoers,dc=example,dc=com
objectClass: top
objectClass: sudoRole
cn: user
sudoCommand: /bin/grep
sudoHost: ALL
sudoOption: !authenticate
sudoRunAsUser: ALL
sudoUser: user

```

编辑好上述内容后，使用如下方式将其导入 LDAP 服务器当中。

```

[root@station103 ~]# ldapadd -x -w redhat -D "cn=Manager,dc=example,dc=com" -f
sudo.ldif
adding new entry "dc=example,dc=com"

adding new entry "ou=sudoers,dc=example,dc=com"

adding new entry "cn=user,ou=sudoers,dc=example,dc=com"

```

如果你觉得命令行操作不方便，也可以使用图形化工具 LDAP Admin 实现，官方地址是 <http://www.ldapadmin.org/>。因为该工具用法简单，我们就不在这里赘述了。

### 13.3.2 客户端配置

接着以 station101.example.com 为例讲解如何在客户端上配置基于 LDAP 服务的 sudo。

#### (1) 安装客户端软件包

```
# yum -y install openldap
```

#### (2) 编辑配置文件

按照如下所示修改配置文件 /etc/sudo-ldap.conf。其中，uri 用于向客户端宣告哪台是 LDAP 服务器，sudoers\_base 则指定了 sudo 配置信息在树形节点中的存储位置。

```

[root@station101 ~]# egrep -v '#|^$' /etc/sudo-ldap.conf
uri ldap://station103.example.com
sudoers_base ou=sudoers,dc=example,dc=com

```

另外还要修改 /etc/nsswitch.conf 文件，这里用于调整 sudo 的验证顺序。也就是说，对于 LDAP 的信息查询，服务器要优先于本地。

```
[root@station101 ~]# grep 'sudoers:' /etc/nsswitch.conf
sudoers:      ldap files
```

如果 LDAP 服务无法连接，此时凡是执行了 `sudo` 的客户端都会一直挂起并等待超时。如下策略设置表示，如遇失败会立刻返回。这样能够有效避免无谓的等待时间。

```
# echo "bind_policy soft" >> /etc/openldap/ldap.conf
```

### (3) 测试执行

首先对比检查，看看 `sudo` 的配置是否生效了。

```
// 没有 sudo 时是无法查询 shadow 文件的
[user@station101 ~]$ grep ^user /etc/shadow
grep: /etc/shadow: Permission denied
// 这是使用 sudo 后的执行结果
[user@station101 ~]$ sudo grep ^user /etc/shadow
user:$5$EtwgEp7R$T6iR6lyKrP8PUdMuoUiG8dXEnVunxAPalOpMmCpxeQ9:17399:0:99999:7:::
```

如果 LDAP 服务连接失败，则会切换到本地验证，即检查 `/etc/sudoers` 里面的配置情况。如果本地验证失败，`sudo` 最终将返回失败。就像下面所示的这样。

```
[user@station101 ~]$ sudo grep ^user /etc/shadow
sudo: ldap_sasl_bind_s(): Can't contact LDAP server
[sudo] password for user:
```

## 13.3.3 关于 LDAP 超时和连接数限制的问题

如果 LDAP 服务不仅仅用于 `sudo`，还启用了账户密码的集中验证，客户端需要安装软件包 `nss-pam-ldapd` 和 `pam_ldap`。这时客户端和服务端之间的会话是长连接，不能主动释放。如果连接数或者文件打开数达到 `ulimit` 的限制，`slapd` 服务就会 hang 住。

要解决这个问题，可按照如下方式编辑客户端的配置文件。通过降低空闲超时让客户端尽快释放连接，以减少在线的并发数量。

```
[root@station101 ~]# egrep '^bind_|^time|^idle' /etc/pam_ldap.conf
timelimit 5          // 查询等待的时间
bind_timelimit 5     // 登录等待的时间
bind_policy soft
idle_timelimit 5     // 空闲时间
[root@station101 ~]# egrep '^bind_|^time|^idle' /etc/nslcd.conf
bind_timelimit 5
timelimit 5
idle_timelimit 5
```

## 13.4 密码学与数字证书

在很久以前，原始氏族的生产力低下，使得人类时常受到自然灾害和猛兽的侵袭。人们依靠合作与互助来保证群体的生存。在氏族内部中，人们相互之间是平等的，对一切生



产资料实行平均分配的制度。随着生产力的不断发展,逐渐产生了私有制。私有制的诞生,标志着原始氏族平等的社会关系发生了转变。人类有了不可与别人分享的私有化财产,不仅仅是金银珠宝这些物质财富,同时也包括他们内心的想法。今天我们管这些叫作隐私。为了维系彼此相互之间的合作关系,人们首先需要建立信任。

建立信任必须要回答两个问题——信任谁?如何建立信任?信任对象按照种类可以划分为物件(信物)、人(担保人)以及组织(权威机构)三大类。而建立信任的手段有“锁”“符”“印”三种。

□ 锁——是一种封印工具。人们将财物或密信放置到一个坚固的密闭容器中,然后用锁将容器入口封住,从而形成一种保护。除了锁的主人可以用钥匙打开以外,其他人即便得到容器也无从下手。自从私有制诞生以来,人与锁之间的信任关系就从未改变过。

□ 符——是一种身份的象征。例如,将物品一分为二,俩人各执一半,在需要确认对方身份时,可以拿出来拼合对证。古人有个“破镜重圆”的典故,说的就是一对夫妻失散多年,愣是凭着一分为二的镜子重新相认了。符象征着一个人的身份,今天的符演变成了身份证。

□ 印——具有公信力和法律效力。在古代,做皇帝要有传国玉玺,当官的要有官印。这说明印章可以证实行为的合法性,同时也起到了证据保全的作用。

直至今日,网络时代使信息数据化、虚拟化,人们对于信息安全的需求更加迫切。如何在网络时代构建新的信任关系,保证我们的信息安全呢?我的理解可以概括为如下八个字。

□ 真实——指信息来源的真实性,也就是身份认证。

□ 准确——指信息内容的准确性,防止他人篡改发送人原本要表达的意思。

□ 机密——指保证信息内容不被第三方知晓。

□ 保全——指对信息内容进行取证保全,防止干系人抵赖其曾经的行为。

要实现这八个字,离不开密码学与数字证书的支撑。在我们的日常生活中,不论是网络支付还是电子办公,这项技术已经广泛应用于数据加密、身份认证、电子签名等诸多商业领域。有一点可以肯定的是,数字证书技术和我们的关系越来越紧密,我们经常也会遇到有关于数字证书的一些问题。因此本节将为大家介绍一些相关的基础知识。

### 13.4.1 密码学技术

所谓的加密,就是指将信息(明文)变成一堆无法理解的乱码(密文)的全过程。加密过程中涉及两个核心元素。一个是密钥,我们可以将它看作解锁的钥匙。密钥在加解密的过程中,作为参数和明文一同参与了运算。另外一个算法,也就是加解密的具体步骤。密码算法大致可分为三类。我在介绍它们之前,需要先声明如下这些符号定义,以方便我们后面的讲解。



- 明文  $P$  (Plain Text);
- 密文  $C$  (Cipher Text);
- 密钥  $k$  (Key);
- 函数  $f$  (Function);
- 逆函数  $f^{-1}$ ;
- 发送者 Alice;
- 接收者 Bob;
- 劫持者 Eric。

### 1. 对称加密算法

顾名思义，对称加密是指加解密时使用同一把密钥。其过程可按如下方式来表示。常见的对称加密算法有 DES、RC、AES、IDEA 和 Blowfish 等。

加密： $C=f(k, P)$

解密： $P=f^{-1}(k, C)$

对称加密算法的难题是密钥的保护。在密文传递的过程中，需要考虑如何将密钥安全地送达接收方。这是一个密钥如何交换的问题。如果 Alice 和 Bob 两人彼此相识，且在发送消息前相互见过面，双方可在会面时，悄悄约定以后通信所使用的共享密钥。这种方式称为带外交换，将密文和密钥拆分，以不同的渠道发送出来。如果两个人根本没见过面，在基于不安全的网络上，Alice 和 Bob 都很担心密钥在传递的过程中会被劫持者 Eric 截获。此时，应该如何实现共享密钥的协商呢？

两位科学家 Diffie 和 Hellman 提出了一种很巧妙的协商算法。首先公开一个大素数  $q$  和整数  $a$ ， $a$  是  $q$  的原根。然后，Alice 和 Bob 各自分别生成一个随机数  $X$  和  $Y$ ，使得  $X < q$ ， $Y < q$ 。接下来，两个人据此分别计算自己的中间密钥。

Alice 的中间密钥为  $a^x \bmod q$ ，简称  $A$ ；

Bob 的中间密钥为  $a^y \bmod q$ ，简称  $B$ 。

之后，两个人分别将自己的密钥发给对方，通过下面这个方式生成最终的共享密钥。

Alice 得到了  $a^y \bmod q$ ，计算  $B^x$ ，即  $a^{xy} \bmod q$ ；

Bob 得到了  $a^x \bmod q$ ，计算  $A^y$ ，即  $a^{xy} \bmod q$ 。

最终，我们发现两个人用自己的随机数运算了两次，得到了相同的结果。从始至终， $X$  和  $Y$  都没有离开过 Alice 和 Bob。Eric 即便截获了中间密钥  $A$  和  $B$  也没有任何用处，因为  $\bmod$  求模运算是不可逆的，他没法从中拆解出  $X$  和  $Y$  来，也就不能推出最终的共享密钥。这就是著名的 DH 密钥协商算法。

### 2. 非对称加密算法

非对称加密算法也称公钥加密算法，是指加解密时使用两把不同的密钥。它是由 Diffie

和 Hellman 在 1976 年的一篇论文——《New Direction in Cryptography》中首次提出来的。它的理论实现基于单项函数。如果一个定义域  $x$ ，在经过某种函数  $f(x)$  计算后得到了值域  $y$ ，但无法通过逆运算  $f^{-1}(y)$  推导出  $x$ ，那么这种函数称为单项函数。常见的非对称加密算法有 RSA、ECC 和 ElGamal 等。

非对称加密算法拥有两把不同的密钥，两把密钥相互都不能推导出对方。使用其中一把密钥进行加密后，需要另外一把密钥进行解密。具备这种特点的函数算法也称陷门单项函数。两把密钥一把用于公开发布，我们将其称之为公钥；另外一把被称为私钥，需要由 Owner 妥善保存。

### 3. 哈希算法

哈希算法并不是一种严格意义上的加密算法，因为它是不可逆的。另外，不管被 Hash 处理的明文内容有多长，其运算结果始终是一个固定长度的值，而且明文内容被稍有变动都会影响到最后的结果。哪怕只是修改了一个字节，两个 Hash 值都会大相径庭。哈希函数的目的就在于，将明文转化为一段固定长度且不可逆的密文，以此作为明文内容的信息摘要。因此，它也被称为单向散列函数。

事实上，我们对此并不陌生。很多软件开发者在发布安装包或源代码时都会提供 MD5 或是 SHA-1 的校验信息，以此证明软件来源的合法性，防止别人从中动手脚（比如捆绑木马、植入病毒程序等）。每个平台上都有 Hash 计算工具，例如 Windows 平台的 WinMD5、Linux 平台的 md5sum 等。软件包下载完后，你可以重新对其进行 Hash 计算。如果计算结果和发布者提供的校验信息一致，则说明该软件包未被篡改，信息源头来自发布人，可以放心使用。

在 RFC 2104 规范中，Hash 函数还与用于确认消息发送方身份的消息认证码（Message Authentication Code, MAC）相结合使用，使得 HMAC 成为 IP 安全中强制实行的 MAC。

不过，Hash 函数存在着一值多解的风险。Hash 的结果是一个固定长度的值，而被 Hash 的明文又可以是任意内容，肯定会出现多个不同内容的 Hash 值结果相等的可能，也就是我们常说的碰撞冲突。

早在 1994 年，Van Oorschot 和 Wiener 就曾经考虑过在散列中暴力搜寻冲突函数（Brute-Force Hash Function）。只不过由于 Hash 值非常大，实际应用中只是在对赌这种碰撞冲突的低概率而已。由于哈希算法的广泛应用，碰撞冲突的可能性会越来越大。像 MD5、SHA-1 这些长度不足的算法，已经被数学家们证明了破解的可能。因此，我们需要更换那些长度更长的新算法。

Linux 系统用户的密码是以加密形式存储在 shadow 文件中的。字段的第二位代表使用的算法种类。1 代表 MD5，2a 代表 Blowfish，5 代表 SHA-256，6 代表 SHA-512。强烈建议用户将算法更换成 SHA-256 或 SHA-512。使用 passwd 命令修改密码时，程序会根据所选算法进行加密。可以使用 authconfig 命令查询或修改当前的加密算法。

执行如下命令，可将你系统的当前算法设置成 SHA-512，并检查是否生效。算法更新后只有再次修改密码时才会生效，shadow 文件中已经存在的密码序列不会变动。如果之前系统采用的是 MD5 算法，请在算法更新后，使用 passwd 命令将所有系统账户的密码重新修改一遍。

```
# authconfig --passalgo=sha512 --update
# authconfig --test | grep hashing
```

### 13.4.2 数据加密与数字签名

综上所述，密码学技术的应用场景包含数据加密和数字签名两种。或是确保数据内容的安全传递，或是确保信息来源的可靠性。

对称加密算法的优点在于运算速度快，但密钥管理的风险很大。而非对称加密算法的优劣则恰恰相反。孤立运用某一种算法并非明智之举，实际应用中的解决方案是综合了两者优势后的产物。下面，我将介绍数据加密和数字签名的实现过程。

#### 1. 数据加密的过程实现

数据加密的全过程如图 13-1 所示。它是基于使用公钥加密共享密钥，并用共享密钥加密明文的方式来实现的。

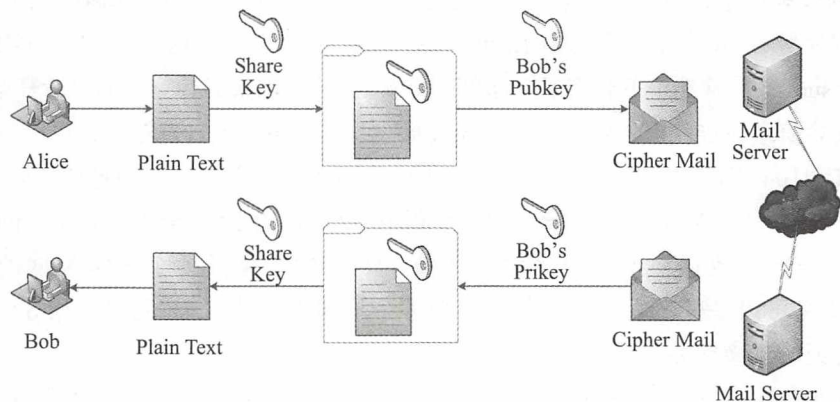


图 13-1 数据加密的过程实现

在进行通信之前，需要通信双方拥有自己的密钥对。公钥用于相互交换，私钥必须妥善保存。假设 Alice 要在互联网上给 Bob 发送一封加密邮件。Alice 需要提前获取 Bob 的公钥，先使用对称加密算法的共享密钥来加密明文信息，然后使用 Bob 的公钥加密共享密钥，最后将密文和加密的共享密钥一同发送。Bob 收到这封密信后，可先用自己的私钥解密拿到共享密钥，然后再用共享密钥解锁明文信息。

因为非对称算法的运算速度慢，如果加密整个消息会严重拖慢执行的时间。对称算法的运算速度快，只是共享密钥的协商和传输是一个难题。使用公钥加密共享密钥，既节省



了加密时间，又保障了共享密钥的传输安全，可谓是一举两得。

那么如何证明这封邮件是安全的呢？我们可以根据之前介绍过的非对称加密算法的特点来论证这一事实。首先，公私钥相互不能推导出对方。其次，公钥加密后只有私钥才能解密。第三，私钥被 Bob 妥善保管，没有其他人截获过 Bob 的私钥。因此，我们认为 Alice 发送的加密信息是安全的，只有持有私钥的 Bob 本人才能阅读这封密信。

## 2. 数字签名的过程实现

数字签名的全过程如图 13-2 所示。它是基于使用公钥加密 Hash 结果，然后接收方自行计算 Hash，并和原值比较的方式来实现的。

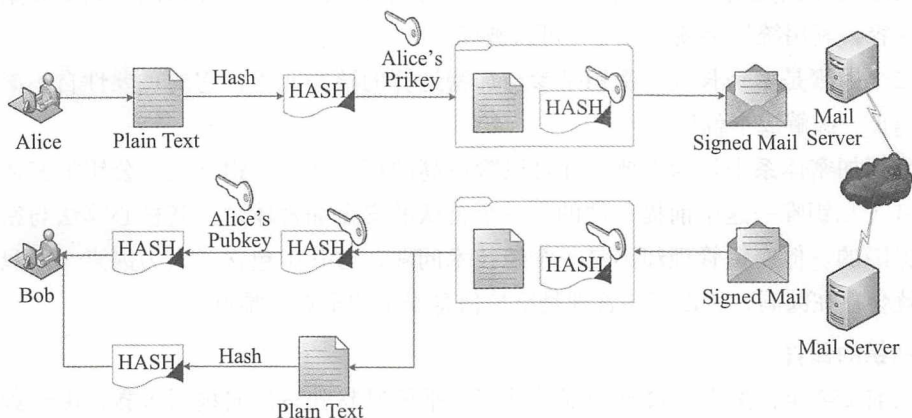


图 13-2 数字签名的过程实现

我们前面讲到的哈希算法，虽然可在一定程度上验证信息来源的可靠性，但是这种验证是不完善的。假设软件发布站点被入侵者 Eric 控制。Eric 在对程序植入恶意代码后，重新计算了 Hash 值，并将这个假的 Hash 值连同篡改程序一同发布出去。如果站点负责人对此并不知情，后续的下载者都将成为受害对象。

数字签名可以有效地解决这一问题。假设 Alice 要在互联网上给 Bob 发布一份电子合同，且 Bob 已经提前获取了 Alice 的公钥。那么，Alice 可先对这份合同内容进行 Hash 并生成摘要信息，然后使用自己的私钥加密这段摘要信息，最后将加密过的摘要信息连同合同一同发送。Bob 收到合同后，一方面使用 Alice 的公钥解密，从而获得了 Alice 计算的摘要信息 A；另一方面在本地对该份合同再做一次 Hash 计算，获得自己计算的摘要信息 B。如果两个信息一致，说明这份合同没有被篡改，否则该合同是无效的。

如果中间人 Eric 截获了合同，篡改并尝试伪造了新的 Hash 值，这种做法没有任何意义。原因是 Eric 在加密 Hash 值时，无法使用 Alice 的私钥。这个 Hash 值即便被加密了，也不可能使用 Alice 的公钥来解密。我们前面已经说了，私钥加密后只有它的公钥才能解密。解密失败也意味着合同是不可信的。

数字签名不但验证了合同来源与内容的可靠性,还具有抗抵赖的法律效力。只要验签通过就说明这份合同内容来自 Alice,且表达了她本人的真实意图,那么 Alice 就要为这份合同的内容负责。

### 13.4.3 公钥加密体系的安全性论述

公钥加密体系的安全性包含算法、密钥长度以及密钥管理三个方面。

一个安全的加密体系,其算法必须是公开的。加密体系的安全性是基于数学难题而非依靠闭环算法来实现,否则,它就没有任何价值了。比如,凯撒密码就是一个典型的反例,它的算法原理是固定的字符代换。一旦得知密文是使用凯撒密码加密的,只要收集足够多的密文内容,利用统计字频和穷举就可以破译。

第二个因素是密钥长度。密钥是参与加密运算的核心元素,它的重要性自不必说。密钥长度越长,破解成本就越高。

而公钥加密体系中最为重要,并且风险最高的环节就是密钥管理。公钥加密体系的安全性是基于私钥唯一这个前提条件的。一个公认的安全加密体系,其核心算法与密钥长度都是有保障的。但密钥管理却不是一个纯技术问题,这其中包含了人为因素。而只要有人的地方就会存在漏洞,所以密钥管理是维护信息安全的重点、难点。

#### 1. 私钥的保管

公元前 258 年,秦军围攻赵国都城邯郸。平原君赵胜写信向魏国求救,由于魏王害怕秦国报复,于是令本已派出的十万大军止步于邺。赵胜的妹夫——魏公子无忌听取了门客侯赢的建议,请魏王的宠姬帮忙盗取了魏王寝宫中用以调动军队的虎符。无忌盗用虎符,命令大军继续前行,以解邯郸之围。

虽说史书中尚未详细记载有关虎符的情况,但可想而知,当时魏军的统帅晋鄙在统领大军时不可能没有兵符。而魏王的虎符也能调动大军,说明至少有两把钥匙可以支配魏国的军队。晋鄙和魏王各持一半,而且虎符还可以用来验证对方的资质与身份。如果将魏王的虎符看作私钥,虎符的丢失让此次假传圣旨的无忌“黑”了魏国十万大军。因为虎符是真的,只要验签通过,无忌发布的所有命令都是有效的。

但那晋鄙也并非凡人,不得不佩服他的安全意识,竟然怀疑虎符是无忌盗取来的。于是,他假借“将在外,君命有所不受”不听调遣。哪承想被无忌身边随行的朱亥一铁锥砸成肉酱,白白地丢了性命。

我讲这个小故事说给大家听,是为了再一次证明私钥保管的重要性。一旦私钥泄露了,所有的安全都将化为灰烬。执行 `ssh-keygen` 命令可以产生一个密钥对。在这个过程中系统会要求用户键入一个 `passphrase`,这个 `passphrase` 是用于保护我们私钥的密码。这是因为私钥不能外流的缘故。我们日常在 PC 上用于支付的网银 U 盾,也有这样一个 `passphrase`。在登录账户或支付款项前,需要验证用户身份,也就是数字签名的全过程。我们已经知道,



签名要调用用户的私钥加密，而私钥是保存在 U 盾中的，用我们的行话说就是——私钥不能出 Key（U 盾设备也叫 USB-Key）。应用程序要调用私钥，必须经用户授权，也就是要输入 passphrase。假如你的 U 盾丢失了，在不知道 passphrase 的情况下是无法盗刷卡内现金的。而我们使用公钥登录 SSH 时，这个 passphrase 通常都为空，那么身份验证这一层的安全性就只能取决于系统自身的安全程度了。

## 2. 公钥的发布

除了私钥的妥善保管，如何确保一个公钥的合法来源同等重要。假设 Alice 和 Bob 之间根本就没有可信的传递公钥的方式，那么 Alice 又该如何确定自己拿到的那个公钥确实是 Bob 本人的呢？如果中间人 Eric 同时欺骗了 Alice 和 Bob，两个人都将 Eric 的公钥当作对方的公钥来使用，后果不堪设想。误用了假的公钥和私钥泄露是同样危险的。

### 13.4.4 数字证书是什么

如何才能确保公钥的安全发布呢？如果能有一个公认的权威机构去负责，那真是再好不过了。权威机构要发布一个可信的公钥，必须完成如下几件事。

- 妥善存储并管理用户的公钥及其相关信息；
- 将用户信息和公钥封装进一个容器；
- 在容器内写入颁发者信息并对所有内容进行签名。

这个权威机构被称为 CA，而这个容器就叫做数字证书。我们看到，数字证书中不但包含了用户的信息和公钥，还有颁发者 CA 的信息。这些内容都是 CA 用私钥签过名的。用户在得到数字证书后，可以使用上级 CA 证书中的公钥来验签，并确认证书来源的合法性。

看上去，数字证书很像身份证对么？但是，我们已经了解了公钥加密体系的作用不仅仅是简单的身份识别。它实现了数据加密和数字签名的功能，而且数字签名是具有法律效力的。由此可见，数字证书是“锁 + 符 + 印”的现代网络版。

### 13.4.5 数字证书是怎么产生的

数字证书按照载体类型可分为软、硬两种。所谓软证书是指以文件形式存放的证书。由于文件容易复制传播，所以软证书的安全性比较差。而硬证书则被存储在硬件设备里。证书的类型和用法不同，存放的设备也不同。设备证书通常会被烧录在硬件的固件里，而用户证书通常保存在 USB 接口设备中，也就是我们常说的 U 盾或 USB-Key。

生成一张数字证书的步骤包括生成密钥对、生成证书请求和证书签发。私钥不外露是生成密钥对的基本原则，私钥在哪里产生的就要存储在哪里。生成私钥时，为了保护私钥不被非法使用，必须设定 passphrase。只有软证书可以置空 passphrase，硬证书通常是不允许这样做的，甚至还会进行弱口令检查。

如果想要在 Linux 系统上产生一组密钥对，可以执行如下这条命令。



```
# openssl genrsa -rsa -out my.key
```

有了密钥对以后，申请人需要填写用户信息，然后使用自己的私钥对信息进行签名，并生成证书请求，最后将证书请求发送给证书权威机构。用户信息中的常见项目包括了 C（国家）、S（省）、L（地区）、O（组织机构）、OU（下级部门）、CN（通用名称）、E-mail 等。其中 OU 可以有多个，毕竟组织机构下可能会有多个层级。CN 通常用来标识和分辨一个证书的主体，也就是 Owner。用户证书的 CN 多为使用者名称或 ID 编号，设备证书的 CN 多为设备名称或 SN，站点证书则为域名或 IP 地址。

我们看到证书信息是由一组元素组成的。组成的元素越多，其辨识性就越强。也即是唯一性越强，产生冲突混淆的可能性也就越小。如果一组信息下来，能够完全辨识出一张证书的信息就称作 DN（可辨识名称）。由于 CN 本身也很容易辨识证书主体，所以它也经常作为 RDN（相对可辨识名称）而存在。

我们来举个例子。假设用户张三有一张数字证书，证书信息如下。

- ❑ CN = zhangsan;
- ❑ OU = IT;
- ❑ O = Example, Inc.;
- ❑ L = Daxing;
- ❑ S = Beijing;
- ❑ C = CN。

CN 标识了张三的名字，我们可以将 CN 当作主语。在 IT 部门里可能只有一个叫张三的，那 CN 就是唯一的，仅靠 CN 就可以辨识出证书主体，所以此时的 CN 就是一个典型的 RDN。但证书机构的管理范围很大，比如省级 CA 管辖着一个省的用户证书，叫张三的人多得是，不可能仅靠 CN 来区别，那么就需要更多的信息来修饰限定 CN。这些内容可以看作定语，定语越多，出现重名的可能性就越少。DN 就表示在 CA 的全局中名称都是唯一的，而 RDN 则表示在某个范围内名称是唯一的。执行如下命令，可用于生成一个证书请求。

```
# openssl req -new -key my.key -out my.csr
```

证书请求应该由权威机构的 CA 服务器签发。自己请求、自己签发的证书叫自签名证书。执行如下命令，可完成一张证书的签发。

```
# openssl ca -in my.csr -out my.crt
```

### 13.4.6 数字证书是怎么验证的

证书错误警告很常见。当我们访问服务器带外管理的 Web 页面时，就容易出现这类错误。下面我们来分析一下出错的原因。

数字证书会从信任锚、有效期、CN 项三个方面去验证。

### (1) 信任锚

信任锚是一种标准的称呼，俗称证书链。它描述了数字证书信任关系中的一个完整的路径。每张证书都有它上一级的颁发者，从顶级颁发者（也称根 CA）一直到最下面的一张具体的数字证书，形成一条完整的信任关系链。

以身份证为例。我们的身份证是在街道派出所办理的，派出所可以看作一个下级 CA，派出所归属市公安局管理，市局归属省厅管理，省厅归属公安部管理。我们看到，一条完整的信任锚体系就如同下面所示的关系一样。

公安部 -> 省厅 -> 市局 -> 派出所 -> 张三的身份证

这条信任关系链有四级机构，张三的证书就是通过这样一种关系签发下来的。如果缺少了任何一个环节，都称为证书链不完整。验证信任锚需要用到上级 CA 证书中的公钥，因为每张证书里的信息都是经过上级 CA 私钥签名的。简单地说，就是用上级去验证下级，验签通过证明下级证书是可信的。只有每一级证书都能验签通过，证书链才是完整的，所有的内容才可信。否则，系统就会返回错误。

好，现在我们有一个疑问。既然是上级验证下级，那么位于最顶级的根 CA 是如何验证的呢？根 CA 的证书简称为根证书。由于根 CA 位于整个信任锚的最顶端，根证书属于自签名证书，没有办法验证，所以系统对于根证书的导入十分慎重。每当导入一张根证书时，系统都会提醒用户，确认导入行为是安全的。如果根证书本身造假，那么其验证通过的证书就都是假的。

因此，如何确保根证书的权威发布是整个数字证书信任体系的最初源头。全球知名的 CA 机构包括 VeriSign、Trustwave、GeoTrust、GTE、GlobalSign、Thawte 等。这些机构的根证书都会事先内置在操作系统里，由各大操作系统厂商保证证书的权威发布，无须用户手动导入，以防止根证书造假。

### (2) 有效期

身份证过期后是无法使用的，数字证书也一样，这个不必过多解释。数字证书上包括了签发日期与过期时间。如果当前系统时间不在有效期内，则会返回错误。

### (3) CN

如果证件持有者的容貌与证件照不符，我们就有理由怀疑证件是假的。系统在进行身份验证时也要比对证书信息。我们已经讲过，由于容易辨识的缘故，关键信息大多写在 CN 中。例如，站点证书的 CN 项通常是域名或 IP 地址。如果浏览器中地址那一部分的信息和 CN 不同，系统就会阻止访问并返回错误。

#### 注意：

证书验证失败后会阻止访问，并返回错误类型来警示用户。用户可在确认警告后强制继续访问。当然，随后产生的所有风险将由用户自负。

## 13.5 人为因素

线上系统发生故障后，业务量会不同程度地下跌，造成经济损失。发生故障的原因我总结了如下几种。

- ❑ 架构容量不足以支撑现有业务的增量；
- ❑ 程序逻辑不严谨导致的异常；
- ❑ 硬件故障；
- ❑ 人为因素。

尽管引发故障的原因多种多样，但有一点不能否认，有相当一部分故障是人为造成的。我们就拿“双十一”期间各大电商的大促活动来说，这个时间的业务量会比平时高得多。但是在此期间，我们很难看到哪个平台出现过重大的事故。因为在大促之前就封网了，不存在任何的线上操作，这段时间反而很安全。只要你的容量撑得住，演练和巡检做到位，就不用担心。倒是封网前的切换和日常上线最容易出问题。

因此，不管你的团队技术水平有多高，管理机制多么完善，也永远不能忽视人为因素给安全生产带来的影响。

### 13.5.1 运维红线

运维红线是运维工作的操作底线，它既保障了线上环境的安全，同时也在维护运维人员的自身利益。其实叫运维红线也好，叫安全规范也罢，本质上没有什么区别，只是红线的语气更强，警示任何人不得触碰。作为一个衡量操作行为规范的尺度，红线的制定要有法理依据，同时应形成文档并公告给所有的人。否则，你就很难拒绝那些不合理的业务需求。发布运维红线，也是为了保证我们的安全行为能够做到有法可依、执法必严。

下面，我们就举几个运维红线的例子。

#### 1. 严禁私自分配资源或变更线上配置

任何资源分配或者线上变更都应该经过工单系统，不能仅靠发邮件或以私下协商的方式执行任何操作。

#### 2. 严禁密码明文出现在命令行中

使用 VBS 等自动化登录脚本的这种行为在 SecureCRT 上特别泛滥。很多脚本的逻辑并不严谨，在一些异常条件下，会将密码当成命令输入命令行中。由于操作命令是记入日志的，这等于泄露了密码。与此类似的行为还有：使用 echo 修改密码，在 ipmitool 操作中以明文方式发送密码等。

#### 3. 回退机制和灰度发布

重大的线上操作必须要有完整的回退机制，以保证执行失败后可以恢复到原始状态。



在正式操作前，还要经历充分、严格的先期测试。另外，对于线上的批量操作，一定要以灰度发布的方式进行。先试着操作一个节点，如果没问题可以再尝试操作一组节点，按照递增的方式来执行，以防因异常而引发全局故障。

#### 4. 严禁越权操作

前几章里我们不止一次地强调，SE 对业务熟悉程度不高。所以，除非 SSH 的管理通道失效，严禁 SE 执行任何启停应用进程或重启、关闭主机的操作。我个人比较反对 SE 参与业务系统的变更工作，除非你的团队没有 PE 或者 DBA。SE 是系统层面的负责人不假，但让其参与业务系统的变更是有风险的。因为他们对业务不熟悉，要关闭一台业务主机，需要业务方提供 IP 地址，如果因为给错了地址而发生事故，那该由谁来负责呢？对于业务变更，SE 可以从旁协助，但不要让他们参与任何修改。

#### 5. 严禁泄密

不管是出于什么原因，泄密的性质都是极其恶劣的，后果也非常严重。例如，把密码堂而皇之地放在邮件中来发去；使用明文传输协议登录并上传敏感信息；商业数据以明文方式存储在数据库中；工作文档、产品源码随意地上传到公有云等，这些都属于严重的泄密行为。不管有什么样的理由，都是不能容忍的。

上述这些行为，都是运维工作中绝对不能触碰的红线，每一个运维人都该以此警醒自己。

### 13.5.2 安全操作

既然大多数故障都和人员操作有关，那么接下来，我们再说说安全操作的事儿。

#### 1. 数据中心的安全操作

在数据中心的机房里工作时，一定要小心周围的设备和线缆。尤其是那些采用密闭冷通道的新机房，机柜大多是不装前门的。不谨慎小心的话，很容易中招儿。

静电是电子设备的大敌，老一辈的工程师都自备静电手镯。虽说现代数据中心的机房内大多铺设了防静电地板，但有时设备会移至调试间操作，所以我们最好还是采取一些主动措施为妙，以防万一。

移动设备或更换线缆前，一定要点亮 UID，以防弄错操作对象。工作完成后要及时关闭它，不要给别人造成困扰。除了上架、布线和本地管理之外，其他工作不要在机房内进行。有些人喜欢在机房里进行设备维修或更换配件，东西码放一地，这种坏习惯实在是不得。

#### 2. 线上系统的安全操作

养成良好的操作习惯，在执行一条命令之前，一定要慎重，看清楚再按确认键。否则，随手一个回车下去，后悔就来不及了。

前面我们提到了自动填充工具的危险性。同样，复制粘贴也要慎用。剪贴板有时候非常坑人，我明明复制了条目 A，可内容却没有更新，粘贴时还是之前的历史条目 B。记事本的自动换行功能会将原本的一行命令截断成两行，导致操作失败。

### 3. 误操作是怎么产生的

2017 年，某著名公司曾发生了一次误操作事故。相关部门在事后发布了处理公告，宣称他们针对误操作的原因做了一些技术上的完善，但始终没有提到在人员方面的任何改进。我个人认为，如果他们真的忽视了，恐怕今后这方面的学费还要再补交一次。

有些同行可能会认为，人难免会出现误操作，所以应当依靠技术手段去规避风险。但我认为误操作不是误差，是完全可以避免的。我的反驳理由很充分：如果你所有的资产都存放在你维护的服务器上，你还会觉得误操作是在所难免的吗？产生误操作的原因很简单，归根到底就是事不关己。我们总强调责任心、敬畏心，但实际上误操作事件还是不断地出现。为什么呢？还不是因为和自己的切身利益不挂钩。

误操作就是人为过失，不要给自己找任何借口。我曾经处理过很多测试环境的系统重装工作，大多数是由于用户误执行了诸如 `rm -rf *` 或者 `chmod -R 777` 之类的命令造成的。重装的次数多了，有些用户就采取改名的方式，企图借此来规避问题，但这样做并无太多意义。误操作是人为意识，如果不解决根本问题，当你要执行真正的删除操作时，岂不依然存在风险？

所以，不能仅靠技术手段来避免误操作的风险，关键是要弄清楚产生误操作的根本原因是什么。缺乏责任心我们就不提了。除此之外，急躁和注意力不集中也是引发误操作的主要因素。有些老运维干了十来年，也难免“晚节不保”，多半都是栽在了这上面。

运维工作本来就很辛苦，除了日常工作外，很多应急事件都发生在休息时段。由于应急响应要求，运维人员必须在几分钟之内上线进行处理。当事件突发在半夜时分时，人的大脑往往是不清醒的。这会儿最好先去洗把脸，好好清醒一下。不管怎样，故障已经发生了，也不差这一分钟。本来你现在就迷糊，还硬要去上线操作，反而有可能会造成更大的损失，那就非常划不来了。

急躁也是运维工作的大忌，通常是因疲于处理并发事件而引起的。就算并发事件再多再急，我的解决方式也是将其排队，一个一个地来。我不是处理器，没有并发的能力。如果你要同时干多件事，就要打开多个 Console。这样做很危险，弄不好就会把 A 主机的命令执行到 B 主机上去。还有，在执行关键操作时，不要接电话或者聊 IM，被人催促时也不要慌张。有一些经验少的新同学，就怕领导或同事的催问。活还没干，自己心里就先乱了。要记住，谁大也大不过线上系统的安全。你一旦分心，就有可能铸成大错。

### 13.5.3 运维工作中的常见问题

除了上述这些问题之外，我们日常工作中还会犯一些不起眼的“小错误”，如果不加以



重视，后果也是很严重的。

### 1. 应急响应

线上的紧急故障处理，一定是以恢复业务为主要目标的。除非你的生产系统全面瘫痪，否则不要把时间消耗在故障排查上面。服务器 hang 死就赶紧重启，并做好业务切换的准备。如果节点重启后还是无法恢复，就立刻切走，不要耽搁时间。待业务恢复后再分析原因也不迟。要知道，从你接电话到上线少说要花三五分钟的时间，再加上各种操作，怎么着也得将近 10 分钟了。哪有工夫等你在那儿慢慢琢磨？高等级的 SLA 一年能有几个 10 分钟给你挥霍？

发生问题不能乱，要各司其职，各尽其责。有些部门一出故障，恨不得全公司的领导都打电话过来问一遍情况。底下人光接电话了，根本没时间处理问题。即使是找到问题，领导不拍板，谁也不敢动。问东问西的，时间全耗过去了。各个团队的分工要明确。技术部门的人，你就让他专心地处理故障；业务部门负责关注线上的状态；服务团队负责通知用户；各部门领导负责沟通协调并关注事件的进展。要做好应急响应，平时必须加强有针对性的演练，减少无谓的沟通和决策时间。

### 2. 要留退路

在你要重启服务器或者修改网络、SSH 等和远程访问有关的配置之前，请先通过 OOB 登录到该节点的管理控制台上。如果 OOB 不能访问，请先解决 OOB 的问题，可以在系统内使用 IPMI 的 open 接口来重置带外管理卡。确保备用通道是活的。别到时候业务系统出了问题，连带外也登录不了，那就太尴尬了。

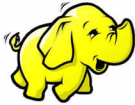
### 3. 提前准备

每逢节假日，都是组内成员轮流值班。如果第二天轮到自已的班，请头天晚上提前打开计算机并连上网络。不要等出了问题以后，再临时开机上线。这是我长期运维工作积累的一个小经验。因为运维团队平时的邮件就比较多，长假期间积累的量会更多。如果你不提前开机，头几分钟内肯定是无法查收最新消息的。这个问题在实时同步的邮件服务上尤为明显。此外，诸如文档索引、病毒库、补丁升级等也是同理。假如你要搜索个文档，可偏在这时，系统告知你正在建立索引需要耐心等待较长时间，以及杀毒软件好久没有检查系统了，各种补丁要更新升级了，等等。本来就够烦的了，这些东西还跳出来给你捣乱，并抢占系统资源，你又怎么能够安心地处理问题呢？

### 4. 测试设备的信息泄密

测试也是运维人员的一项工作。SE 的主要测试对象就是服务器了。测试设备最后是要归还的，有些测试人员不经意间，可能就把公司的一些信息给泄露出去了。我就曾在某台测试机的 OOB 上看到了某公司 DNS Server 的内部地址。之所以我知道是哪一家公司，是因为测试人员不但在 OOB 上配置了 DNS Server，而且设置了邮件告警功能。这些信息都没





有擦掉，所以一看邮箱域名就清楚了。

不单是带外管理，磁盘内的系统信息更敏感。比如，大家敲个 `last` 命令，自己的地址首先就暴露了。`yum repo` 文件能够交代的東西更多，直接就告诉人家你们的管理区地址段是多少，以及 `yum` 源服务器是谁。如果 `rsyslog` 是转发的，想知道 `Log Server` 的地址并不难。`resolv.conf` 坦白了 `DNS Server` 的地址，系统日志则描述了与其交互的服务器有哪些。最重要的是你的 `shadow` 文件，但愿这台测试机的账户没有使用过生产环境的密码。

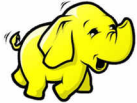
在归还测试设备之前，一定要清除相关的敏感信息，不管是 OOB 还是 OS 都不能放过。擦除 OS 信息时不要简单地使用 `rm -rf *`，那是没有用的。`rm` 命令是按照根目录下的字母顺序来删除的，而 `rm` 自己就位于 `/bin/` 目录下面，你想这样能删掉多少数据？

`dd` 是比较稳妥的，不过在执行 `dd` 之前，最好先擦除掉上述文件的内容。按照 NIS 美国信息安全标准，最少也应当要擦除七次以上才是保险的。擦除信息可使用 `shred` 命令来完成。但前提是必须禁用系统的 `journal`。这一点我们在第 5 章中已经提过了。`dd` 掉系统以后，应进一步清除阵列卡内的信息。

当然，最好的方案是部署一套专门用于测试的环境。谨记不要把任何生产配置写入测试设备中。

## 13.6 本章小结

本章我们围绕操作系统层面，分享了一些安全基础知识，包括密码学基础、数字证书、如何实现 `sudo over LDAP` 以及笔者个人的一些心得体会。下一章我们将会进入一个神秘的领域，聊一聊系统性能校准的话题。



## 第 14 章

# 性能校准

很多人都把 Performance Tuning 理解成性能调优。实际上，这是一个认知上的误区。调是调，优是优，这是两个完全不同的概念。

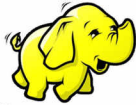
优化的英文是 Optimize。它的实施方式很明确，基本上全是套路。比如增加缓存、提升并发数、关闭无用服务、减少系统层级，等等。优化的效果是可以预见的，在应用了优化方案之后，性能或多或少都会有所改善。

而 Tuning 则不同，我将它翻译成校准。校准的含义是对准与看齐，就像调频收音机上的旋钮一样，要根据实际情况采取不同的应对措施，并没有一套固定有效的调节方式。否则我们就不需要 Tuning 了，一开始就将现成的方案设置成默认值不就好了？

任何校准项目都离不开测试数据的支持，因而整个过程是漫长而玄妙的。引发系统性能问题是如此的复杂，各种影响因素相互耦合交织在一起，涵盖了软硬件多个领域的背景知识，实在是难于理解和分析。何况很多参数之间相互影响，不是一个值就能决定一切的。对于参数的数值调整，其变化趋势也不是单调的。一开始，增加某个数值，可能性能会有所提升。但如果你贪得无厌，继续增加调过了头，性能反而会变得比原先更差。另外，有时候 Tuning 的行为不一定和性能有关，也许只是为了让系统运行得更正常而已（例如与 OOM 相关的调整）。

Performance Tuning 又被人们称为 Black Art（黑色艺术）。Black Art 在西方文化中用来指代类似于巫术的一种行为，意为神秘莫测且难以掌握其精髓。由此看来，这个比喻还真是恰如其分啊。尽管如此神秘，但我并不想过度夸大它的作用。校准的目的只是让系统进入最佳状态，无法成倍地提升性能。Tuning 是一种累积收益，仅仅在一两个节点上看不出特别明显的效果，只有随着节点数量的不断增加，Tuning 的价值才能逐步地呈现出来。

由此可见，Performance Tuning 是真正体现匠人精神的一项艺术。不但需要大量的知识储备和细致入微的观察力，更要有足够的耐心。只有耐得住寂寞，不急功近利，有一颗追寻极致的心，才能立于性能之巅。



## 14.1 队列理论

从宏观角度上讲，性能差体现在两个方面。一个是响应时间。如果一个请求发出后半夭得不到响应，等待时间就会变长。另一个是数据传输。众所周知，吞吐量是反映存储及网络的一个重要的性能指标，数据传输能力越强，完成任务所需的时间就越短。在解决性能问题之前，我们需要先了解一些有关队列理论的知识。所谓“磨刀不误砍柴工”，弄清这些性能影响因素之间的关系后，才能做到有的放矢。

其实，队列理论并不是什么高深的学问，它是运筹学中的知识，说白了就是讲排队的。下面我们就以银行办理业务为例，来说说排队的那些事儿。在讲解之前，我们先看几个概念。

Queue Length( $L$ ) 指的是队列长度。Arrival Rate( $A$ ) 叫作到达率，它表示单位时间内发送的请求数量，反映出一个业务系统的忙碌程度，单位是个/秒。Wait Time( $W$ ) 被称作驻留时间，它指的是完成一个请求后，总共需要等待的时间。我们看到三者之间形成了如下这样一种关系，见公式 14-1。

$$L=A \times W \quad (14-1)$$

从我们的直观感受上说，队列越长，等待时间也就越长。所以，没人会挑最长的队伍来排。从公式上可以看到，减少队列的方法共有两种。一个是降低到达率。但这种限制业务的手段很难实现，除非完成业务拆分或者增加多个队列（比如使用核心数量更多的处理器）。另一个可行性较强的方案就是减少驻留时间。Wait Time 包括 Queue Time( $Q$ ) 和 Service Time( $S$ ) 两部分。前者指的是进程请求资源的时间，后者则是完成请求的服务时间。简而言之，Queue Time 是你在大厅排队时消耗的时间，Service Time 就是你坐在柜台前消耗的时间。而 Service time 又被分为内核态时间与用户态时间两部分。比方说，我要申请一张信用卡，用户填写单据、提交证件等可看作用户态时间，业务员审核、录入、盖章等花费的就是内核态时间。因此，我们可将公式 14-1 展开成公式 14-2 和公式 14-3。

$$L=A \times (Q+S) \quad (14-2)$$

$$L=A \times (Q+T_{\text{sys}}+T_{\text{user}}) \quad (14-3)$$

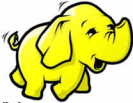
下面这条 time 指令执行后，real 就是总的驻留时间  $W$ ， $Q$  就等于 real 减去 user 和 sys 后剩下的时间。

```
[root@station101 ~]# time netstat -atnup &> /dev/null
```

```
real    0m0.108s
user    0m0.026s
sys     0m0.080s
```

Bandwidth( $B$ ) 反映了一个系统最大的数据传输能力，Troughput( $X$ ) 是指系统的有效传输，Overhead( $O$ ) 是指传输时消耗的部分。公式 14-4 告诉我们，带宽等于吞吐量与开销之





和。由于带宽受硬件因素影响，它的数值是固定不变的，所以提升吞吐量的唯一途径就是减少开销。

$$B=X+O \quad (14-4)$$

公式 14-5 反映的是一个收支平衡的关系。Utilization( $U$ ) 叫作利用率，它用来评估一个系统的工作饱和度。Completion Rate( $C$ ) 称为完成率，它反映了系统的处理能力，即单位时间内可以完成的请求数。

$$U=A/C \quad (14-5)$$

假如，我们将  $A$  看作蓄水池的进水管，将  $C$  看作蓄水池的排水管。当  $A=C$ ，即两者进出水量相等时，说明系统恰好到达了一个平衡状态，水池中的水既不会减少也不会增加。反映在队列当中，就是队列长度不会发生任何变化，既达到了最大工作饱和量，也不会增加系统的压力。

请注意，这个平衡状态在实际生产系统中是不存在的，我们只能通过校准来不断地趋近。到达率作为一个外部因素，如果不进行业务拆分或者硬件升级，它是无法调整的。但是我们可以影响完成率，通过提升完成率，让它尽可能地靠近到达率。当业务员的处理速度提升了，自然就可以降低队列长度。

那么，如何去计算完成率呢？我们就来举一个和 I/O 相关的示例好了。执行命令 `iostat -x 1` 后观察返回的结果。请注意，第一条数据是一个汇总的统计结果，并非实时值。我们从随后的输出里选择一条来说明。

| Device: | rrqm/s | wrqm/s    | r/s  | w/s     | rsec/s | wsec/s     | avgrq-sz | avgqu-sz |
|---------|--------|-----------|------|---------|--------|------------|----------|----------|
| await   | svctm  | %util     |      |         |        |            |          |          |
| sda     | 0.00   | 189328.00 | 0.00 | 3515.00 | 0.00   | 1539128.00 | 437.87   | 140.81   |
| 40.09   | 0.28   | 100.00    |      |         |        |            |          |          |

这里的  $r/s$  和  $w/s$  反映了每秒的读写请求次数，也就是到达率。await 代表 I/O 的平均驻留时间，svctm 则代表一个 I/O 的服务时间。await 和 svctm 的单位都是毫秒。用到达率乘以 await 就是队列长度。而用一秒除以 svctm 则可得到完成率，即公式 14-6。

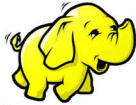
$$C=1/S \quad (14-6)$$

根据公式 14-6，我们可以计算出队列长度和完成率。

```
// 队列长度的计算 (个)
[root@station103 ~]# echo 'scale=2; ( 0.00 + 3515.00 ) * 40.09 / 1000'|bc
140.91

// 完成率的计算 (个 / 秒)
[root@station101 ~]# echo 'scale=2; 1 * 1000 / 0.28'|bc
3571.42
```

然后，我们用得到的完成率和到达率做对比，如果  $A>C$ ，说明  $U>100\%$ ，也就是说队列已经饱和了。根据公式 14-5 和公式 14-6，我们又可以推导出公式 14-7。这样，我们就



可以直接用到达率乘以服务时间来判定利用率是否已经饱和。

$$U=A \times S \text{ (当 } A=C \text{ 时, } U=1; \text{ 当 } A>C \text{ 时, } U>1; \text{ 当 } A<C \text{ 时, } U<1) \quad (14-7)$$

根据公式 14-7, 我们可以进一步计算出利用率。

```
// 利用率的计算
[root@station103 ~]# echo 'scale=2; 3515.00 * 0.28 / 1000'|bc
.98
```

吞吐量也可以在这里计算出来, rsec/s 和 wsec/s 表示完成的扇区数。如果利用率达到饱和, 说明 I/O 负载过高, 通常这种情况下, 吞吐量的数据都不太好看。改善的方式是合并 I/O 操作, 降低磁盘的访问次数。

```
// 吞吐量的计算 (MiB/s)
[root@station101 ~]# echo 'scale=2; ( 0.00 + 1539128.00 ) * 512 / 1024 / 1024'|bc
751.52
```

最后我们总结一下提升系统性能的方法。

第一, 减少驻留时间  $W$ 。可以从下几个方面来入手: 调整调度器算法和参数, 减少无谓的系统调用和中断, 增加 Cache 的命中率, 等等。

第二, 降低开销  $O$ 。例如, 使用 UDP 来替代 TCP。

第三, 提升完成率  $C$ 。例如, 减少访问次数、合并 I/O 操作等。

这些调整方案的内容, 我们会在后面的章节中详细展开。

## 14.2 CPU

“为什么我们选用的 CPU 是低端系列? 你看人家某某部门用的就是……”

“我们的 CPU 占用率很高, 性能太差了! 我们要升级, 因为……”

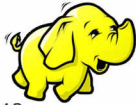
这些话是一些用户抱怨硬件配置的开场白。然而, 当你观察监控数据时却发现, 就是这些所谓的“低端”处理器, 它们在大部分时间里基本上处于空闲的状态。遇到这样的“性能问题”, 有时真是令人哭笑不得。CPU 型号的高低并不代表整机的性能, 而 CPU 的高占用率也不见得就一定存在性能瓶颈。

首先, CPU 占用率的高与低是相对的。在双节点集群服务中, 占用率峰值达到 50% 就是很严重的问题了。但如果是分布式服务, 节点数量足够多的话, 80% 的占用率也很正常。

其次我们要看占用率高在哪里。如果 CPU 资源消耗在了正常的业务上, 只要不存在资源过载的风险, 这难道不是一件好事吗? 这说明我们的生意兴隆, 为什么还要抱怨呢? 资源利用率低反而才是最大的浪费, 成本控制不住, 我们的商业利润又从哪里来?

### 14.2.1 来自内核态的资源消耗

对于技术的深入探索是一件很有意思的事情。同样一个命令, 随着认知深度的不同,



观察点也会有所变化。就拿 CPU 资源的使用情况来说，初学者只关注 idle，而行业老手能看到的東西會更多。

Linux 系統的空間被分成內核空間和用戶空間兩部分，執行級別也分為內核態和用戶態，且兩個空間彼此隔離。用戶進程只能訪問屬於自己的空間，它不能讀取或修改內核空間的數據，也不能執行內核空間的代碼。如果用戶進程要執行某些會影響整個系統的操作（比如操作 I/O 設備），它要通過系統調用向內核發出請求。如果內核確認其有權這麼做，會代表用戶進程完成相關操作。在此期間，內核可以訪問用戶進程的地址空間。系統調用完成後再返回用戶態。

除了系統調用以外，另外一種內核態的使用是中斷處理。它是一種信號，發送者可以是硬件，也可以是軟件，典型的中斷事件就是鍵盤的敲擊。我們在 top 中看到的 hi 和 si 就是硬中斷和軟中斷。當中斷發生時，意味著這裡有個緊急工作要先處理一下，正在運行的進程會被內核暫停服務，待中斷處理完後才能繼續。中斷處理可不能磨磨唧唧的，必須快速了結。要是在處理期間再來一個中斷就麻煩了，那會導致系統死鎖。但是，我們又不可能在處理一個中斷時，禁止其他中斷的到來，因為很多中斷是必需的。為了避免這個問題，Linux 將中斷分成上、下兩個部分。那些對時間非常敏感、與硬件相關的、不能被其他中斷打斷的工作會被放置在上半部分，其餘的則會被放置在下半部分。當一個中斷產生時，先處理緊急的上半部分，剩下的工作可以延遲，待到合適的時機再去完成。這樣就提高了系統的響應能力和並發能力。中斷執行是在內核態，如果此時進程正運行在用戶態，也會發生切換。但與系統調用不同的是，內核無權訪問用戶進程的地址空間。

CPU 的大部分時間都在執行用戶空間的代碼，也就是說一個進程在它的生命週期中，大多數時間是處於用戶態的。當進程進入內核態時，它所消耗的资源就會被記錄成內核態的资源消耗。時間片應當尽可能地為用戶態服務，但這並不意味著 CPU 的內核態消耗就是“不務正業”。進程在內核態中的行為非常重要，而且也是必需的，因此资源消耗在所難免。但在追求一個性能良好的系統環境時，不合理的、過度的內核態損耗是不應該出現的。發生這種問題，往往是不嚴謹的程序執行和錯誤配置造成的。top 命令用來查詢 CPU 资源耗費在內核態（sy）和用戶態（us）的占比情況。time 命令則可以獲取一個程序運行時分別在內核態（sys）和用戶態（user）所花費的時間。

除了系統調用和中斷處理，搶占式多任務也是消耗內核態资源的一個主要因素。現代絕大多數操作系統都可以“同時”運行若干進程，這種多任務處理使得我們以為計算機可以並行處理多個操作。其實，真正意義上的並行是基於多處理器而言的，單個處理器根本不可能實現並行。常言道，心無二用，右手畫圓，左手畫方，則不能成規矩。因為人的注意力只能集中在一件事情上。周伯通那雙手互搏之術卻偏偏要人心為二用，一神守內，一神游外，雙手使出不同的武功招數。這一心二用之法，竟是使心神在兩件事之間不斷地快速切換。當切換頻率足夠快時，自然呈現出一種“偽並行”的狀態。



CPU 的运行速度极快，因此它将时间切分成无数细小的时间片，分发给所有的进程。进程依据重要程度，其所获取的时间片大小也不尽相同。每个进程依靠时间片来获取 CPU 的计算处理服务。当时间片花完了，内核会从进程收回控制权，转而服务另一个进程。而当前进程的运行时环境（即 CPU 寄存器和页表等内容）都会被保存起来，等到该进程在下一轮执行时，再将其运行时环境复原。这样一个进程切换的过程被称为 Context Switch（上下文切换）。在一个进程数超多的系统环境下，上下文切换会非常频繁，这意味着时间片被大量损耗，尤其是在进程游走于多个核心之间的时候，这种跨核心切换对系统性能的负面影响更大。通过观察 `vmstat` 命令返回的 `cs`，可知每秒上下文切换的数量。

```
[root@station103 ~]# vmstat 1 3
procs -----memory----- --swap-- -----io----- --system-- -----cpu-----
r  b  swpd   free   buff   cache  si  so  bi   bo    in    cs  us sy  id wa st
2  0  0    15007224 391260 23670320 0  0  0    4    1     1  0  0 100  0  0
0  0  0    15007216 391260 23670320 0  0  0   124 2505 5595  0  0 100  0  0
0  0  0    15009680 391260 23670320 0  0  0    40 2536 5874  0  0 100  0  0
```

看来，用户态与内核态之间以及众多进程之间的频繁切换是导致内核态过度开销的主要元凶。为此，我们有几种不同的方案来应对。

### 1. 减少无谓的系统调用

`strace` 是一个用来跟踪系统调用的程序，它可以跟踪到一个进程产生的系统调用，包括参数、返回值以及执行消耗的时间等。如果你发现一个程序执行后导致 `sys` 资源占用率升高，想要了解它们的最终用途的话，执行 `strace` 就是最好的方式。

下面这个命令可以用来统计一个程序在它的生命周期内发起的系统调用的次数。如果 `calls` 后面有很多 `errors`，则说明这个程序执行了很多无效的调用。当然，这种有问题的程序通常都来自于系统外部。

```
[root@station101 ~]# strace -c cat /dev/sdal > /dev/null
% time      seconds  usecs/call   calls   errors syscall
-----
97.49      0.008678         1     6402         read
2.51      0.000223         0     6400         write
0.00      0.000000         0         4         open
0.00      0.000000         0         6         close
0.00      0.000000         0         5         fstat
0.00      0.000000         0         9         mmap
0.00      0.000000         0         3         mprotect
0.00      0.000000         0         1         munmap
0.00      0.000000         0         3         brk
0.00      0.000000         0         1         1 access
0.00      0.000000         0         1         execve
0.00      0.000000         0         1         arch_prctl
-----
100.00      0.008901         12836         1 total
```

如果你要针对某一类系统调用进行跟踪, 可以使用 `-e` 来指定调用类型。比如通过指定调用类型 `access` 来跟踪 `cat` 指令对文件对象的访问行为。

```
[root@station101 ~]# strace -e trace=access cat /dev/sda1 > /dev/null
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

## 2. 调整中断分配

`irqbalance` 是 Linux 系统用来控制中断请求 (Interrupt Request, IRQ) 分布的服务, 它默认每隔 10 秒会对中断进行一次平均分配。这种方式未必永远都是最优的。当中断发生时, 如果从用户态切换到内核态, 这也是一种上下文切换的形式。所以在多处理器环境下, 可以考虑让特定的中断在特定的核心上运行, 减少或者禁止在某些核心上发生中断, 确保重要的进程可以一直运行而不会被打扰。

`irqbalance` 有两个核心参数。如果你不想招致每隔 10 秒的额外开销, 可设置 `ONE-SHOT=yes`, 则该服务只会运行一次。`IRQBALANCE_BANNED_CPUS` 可以指定在哪些核心上不能发生中断。

执行如下这段代码后, 我们得到了当前系统中断的分配结果。可以看到, 绝大多数中断都位于第一颗物理 CPU 上 (`core_id` 为偶数), 这未必是最好的方案。

```
// 这是一段用于查看不同中断类型 CPU 亲和度的示例代码
#!/bin/bash
for i in `ls /proc/irq|grep -P "[0-9]"|sort -n`
do
    echo IRQ$i: `cat /proc/irq/$i/smp_affinity_list`
done
// 以下是上述代码的执行结果
[root@station103 Documentation]# sh irq.sh
IRQ0: 0-47
IRQ1: 0-31
IRQ2: 0-47
IRQ3: 0-31
...
IRQ10: 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
IRQ11: 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
IRQ12: 0-31
IRQ13: 0-31
...
IRQ115: 8,24
IRQ116: 14,30
IRQ117: 10,26
IRQ118: 14,30
IRQ119: 10,26
IRQ120: 10
IRQ121: 12
IRQ122: 2
...
```

```
IRQ136: 6
```

```
IRQ137: 6
```

在充分的测试数据支持下，我们可以通过修改 `smp_affinity` 来手动指定 IRQ 与 CPU 的亲和度。`smp_affinity` 使用 `mask` 来描述它们之间的这种关系，它是一个八进制的数值。`mask` 的计算可以借助 `printf` 语句来完成。

例如，我们要将 IRQ10 绑定到核心 1、3、5、7 上，可以使用如下命令来实现。

```
# printf '%0x' $[ 2**1 + 2**3 + 2**5 + 2**7 ] > /proc/irq/10/smp_affinity
```

### 3. 将进程绑定到核心

默认情况下，进程可以游走于所有可用的核心之间。最好限定进程只在某些核心上面运行，以防止跨核心上下文切换。绑定核心的办法有很多，Linux 至少提供了三种方案。

`taskset` 命令提供了一种调整进程与 CPU 亲和度的方法，`taskset` 既可以在进程启动前完成亲和度设置，也可以对正在运行的进程的亲和度进行调整。同样，亲和度关系也是使用 `mask` 来描述的。

```
// 查看当前进程 CPU 亲和度的 mask 值
# taskset -p <PID>
// 查看当前进程 CPU 亲和度的 mask 值所对应的核心
# taskset -pc <PID>
// 在启动一个进程之前绑定核心
# taskset -c <CPU_LIST> <COMMAND>
// 调整一个已存在进程的 CPU 亲和度
# taskset -p <MASK> <PID>
// 下面是一个将 sleep 进程的 CPU 亲和度从 31 调整成 0、1、5、8 的例子。
[root@station101 ~]# taskset -c 31 sleep 100000 &
[1] 36562
[root@station101 ~]#
[root@station101 ~]# taskset -p 36562
pid 36562's current affinity mask: 80000000
[root@station101 ~]#
[root@station101 ~]# taskset -pc 36562
pid 36562's current affinity list: 31
[root@station101 ~]#
[root@station101 ~]# printf '%0x\n' $[ 2**0 + 2**1 + 2**5 + 2**8 ]
123
[root@station101 ~]#
[root@station101 ~]# taskset -p 123 36562
pid 36562's current affinity mask: 80000000
pid 36562's new affinity mask: 123
[root@station101 ~]#
[root@station101 ~]# taskset -pc 36562
pid 36562's current affinity list: 0,1,5,8
```

`taskset` 虽然可以指定进程与核心的亲和度，但它无法保证该核心分配到的是本地内存。所以 `numactl` 是推荐替代 `taskset` 的新方案。关于 `numactl` 命令的使用，我会在稍后的章节



中做详细讲解。

cpuset 也可以锁定进程与 CPU 的关系。假设我们要为 qemu 群组用户添加 cgroup 策略, 可仿照如下方法修改配置文件 /etc/cgconfig.conf, 再重启 cgconfig 服务即可。注意: 使用 cpuset.cpus 时要同时启用 cpuset.mems。

```
mount {
    cpuset = /cgroup/cpuset;
    cpuset = /cgroup/cpuset;
    cpu = /cgroup/cpu;
    cpuacct = /cgroup/cpuacct;
    memory = /cgroup/memory;
    devices = /cgroup/devices;
    freezer = /cgroup/freezer;
    net_cls = /cgroup/net_cls;
    blkio = /cgroup/blkio;
}

group kvm {
    perm {
        task {
            uid = qemu;
            gid = qemu;
        }
        admin {
            uid = root;
            gid = root;
        }
    }
    cpuset {
        cpuset.cpus = 31;
        cpuset.mems = 1;
    }
}
```

#### 4. 减少进程数量

事实上, 类似于 ulimit 这种限制进程数量的方式并不科学。我们很难简单粗暴地去拒绝一个正常的业务请求, 尽快完成业务拆分才是最佳途径。不要把多个服务都跑在一个节点上; 对于高并发应用, 要考虑分布式部署; 对于启用多进程的应用, 在配置进程数目时, 要参考处理器的逻辑核心总数, 不宜过度调整。

### 14.2.2 用户态资源占用率高

如果一个 CPU 的用户态占用率居高不下, 是不是就一定是 CPU 的问题呢? 在事情没有调查清楚之前, 这个锅 CPU 可不背。要知道, 应用程序也有不靠谱的。由于 CPU 运算速度要比内存的读写速度快得多, 这种差异会使 CPU 受到内存的拖累。因此, CPU 的内部设置了一小块高速存储区域, 也就是我们常说的 Cache。Cache 的读写速度是内存的几十倍

不止，如果 CPU 能够从 Cache 中获取数据，性能就会有显著的提升。反之，CPU 要的数据不在 Cache 中，就得从内存中读取数据。这就涉及一个 Cache 的命中率问题。为了举例说明，我们先来看下面这两段 C 语言的源代码。

第一段示例源码：

```
[root@station103 ~]# cat cache1.c
#include <stdio.h>
#define row 8192
#define col 8192

int main () {
    int a,b;
    int total=0;
    static int x[row][col];
    for ( a=0; a<row; a++ )
        for ( b=0; b<col; b++ )
            total += x[a][b];
    return total;
}
```

第二段示例源码：

```
[root@station103 ~]# cat cache2.c
#include <stdio.h>
#define row 8192
#define col 8192

int main () {
    int a,b;
    int total=0;
    static int x[row][col];
    for ( b=0; b<col; b++ )
        for ( a=0; a<row; a++ )
            total += x[a][b];
    return total;
}
```

上面这两段代码，读者朋友们也可以尝试着编译后执行。它们都是从一个二维数组中取值，然后不断累加的过程。所不同的是，cache1 是按行取，cache2 是按列取，而系统也是按行来缓存的。程序初始运行时，Cache 是空的。当你取第一行第一列时（即 1-1），它会把 1-2、1-3 和 1-4 都缓存到 Cache 里来。按行读取的话，后面几次的数据就可以从 Cache 中获取。如果按列读取则大不相同，下一次程序需要的数据是第二行第一列（即 2-1），Cache 里的数据是无效的。

低命中率的程序，在运行速度和资源消耗上自然会比较差，可以用 time 命令来证明这一点。

```
// 编译 cache1.c 并使用 time 测试 cache1 运行所花费的时间
```



```
[root@station103 ~]# gcc cachel.c -o cachel
[root@station103 ~]# time ./cachel
```

```
real    0m0.245s
user    0m0.150s
sys     0m0.094s
```

// 编译 cache2.c 并使用 time 测试 cache2 运行所花费的时间

```
[root@station103 ~]# gcc cache2.c -o cache2
[root@station103 ~]# time ./cache2
```

```
real    0m1.422s
user    0m1.335s
sys     0m0.088s
```

valgrind 命令可以测量 Cache 的命中率。当一个程序的执行效率很低时,除了前面介绍过的 strace 以外,我们又多了一个排查问题的好帮手。通过 valgrind 命令我们可以看出,两者主要的差距在于 D1 (一级缓存的数据区) 的命中率,二者之间竟然相差了近 16 倍。从 LL ref 和 LL misses 上看, L3 几乎也没有缓存这些数据,这就是为什么两者运行时间的差距如此之大的根本原因了。

// 测量 cachel 的命中率

```
[root@station103 ~]# valgrind --tool=cachegrind ./cachel
==6097== Cachegrind, a cache and branch-prediction profiler
==6097== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
==6097== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==6097== Command: ./cachel
==6097==
--6097-- warning: Unknown Intel cache config value (0x63), ignoring
--6097-- warning: L3 cache found, using its data for the LL simulation.
==6097==
==6097== I   refs:      738,339,538
==6097== I1  misses:      625
==6097== L1i misses:      622
==6097== I1  miss rate:    0.00%
==6097== L1i miss rate:    0.00%
==6097==
==6097== D   refs:      402,718,652 (402,700,723 rd + 17,929 wr)
==6097== D1  misses:      4,195,249 ( 4,195,010 rd +   239 wr)
==6097== L1d misses:      4,195,174 ( 4,194,943 rd +   231 wr)
==6097== D1  miss rate:    1.0% (    1.0% +   1.3% )
==6097== L1d miss rate:    1.0% (    1.0% +   1.2% )
==6097==
==6097== LL refs:      4,195,874 ( 4,195,635 rd +   239 wr)
==6097== LL misses:      4,195,796 ( 4,195,565 rd +   231 wr)
==6097== LL miss rate:    0.3% (    0.3% +   1.2% )
```

// 测量 cache2 的命中率

```
[root@station103 ~]# valgrind --tool=cachegrind ./cache2
```



```

==6096== Cachelgrind, a cache and branch-prediction profiler
==6096== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
==6096== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==6096== Command: ./cache2
==6096==
--6096-- warning: Unknown Intel cache config value (0x63), ignoring
--6096-- warning: L3 cache found, using its data for the LL simulation.
==6096==
==6096== I   refs:      738,339,538
==6096== I1  misses:      625
==6096== L1i misses:      622
==6096== I1  miss rate:      0.00%
==6096== L1i miss rate:      0.00%
==6096==
==6096== D   refs:      402,718,652 (402,700,723 rd + 17,929 wr)
==6096== D1  misses:      67,109,809 ( 67,109,570 rd +   239 wr)
==6096== L1d misses:      67,109,734 ( 67,109,503 rd +   231 wr)
==6096== D1  miss rate:      16.6% (      16.6% +   1.3% )
==6096== L1d miss rate:      16.6% (      16.6% +   1.2% )
==6096==
==6096== LL refs:      67,110,434 ( 67,110,195 rd +   239 wr)
==6096== LL misses:      67,110,356 ( 67,110,125 rd +   231 wr)
==6096== LL miss rate:      5.8% (      5.8% +   1.2% )

```

### 14.2.3 Cache 与内存的三种映射关系

既然我们提到了 Cache 的命中率，而 Cache 的数据是从内存中获取的。那么，它们之间是怎么关联的呢？好，我来解答一下这个问题。Cache 与内存之间存在着三种映射关系，如图 14-1 所示。

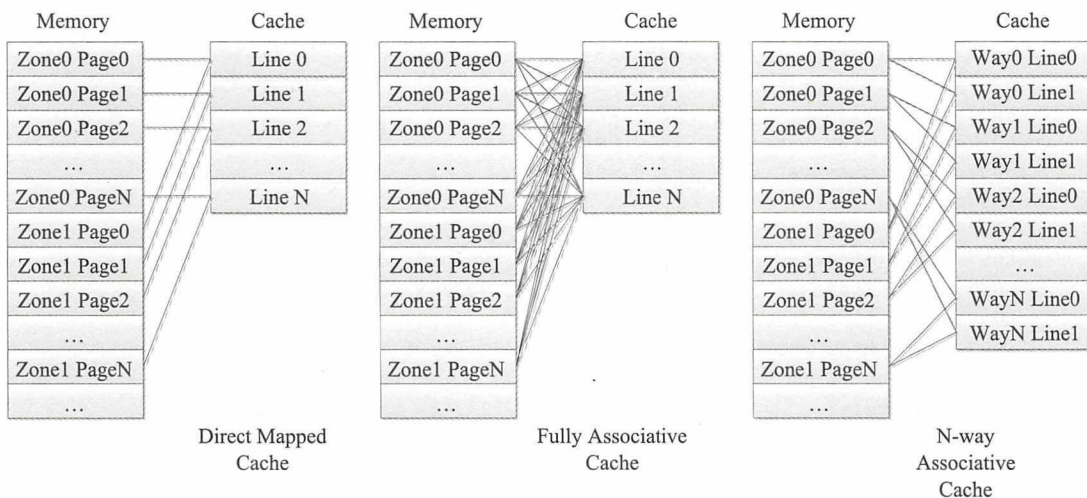


图 14-1 Cache 与内存的三种映射关系

第一种叫作直接映射, Cache 的每一行只能对应内存中特定的块。当然, 内存的容量要比 Cache 大得多。如果要映射所有的内存空间, 必须先将内存划成多个 Zone。在每个 Zone 内, Cache 与内存的地址空间映射是唯一的。如果我们把 Cache 中的行看作座位, 再将内存页看作乘客, 则不同的 Zone 代表了不同班次的乘客, 在同一班次里 (即同一个 Zone 内), 乘客和座位是一对一的关系。就像坐飞机一样, 每人都有属于自己的固定的座位。

直接映射的优点在于对应关系简单、开销小, 但是命中率低。举个例子, 航班 A 的乘客张三选定的座位号为 17G, 航班 B 的乘客李四也选了同一个座位。如果李四要坐这个座位, 张三就得欠身离开。当然, 这种冲突在飞机上是不会发生的, 因为两个人的航班时间不同。但是对于内存访问来说, 这样的矛盾却很常见。如果内存页 Zone1 Page0 要被映射到 Cache 的 Line0 时, 发现里面已经存在内存页 Zone0 Page0, 就要先将 Zone0 Page0 置换出来才行。假使后面的访问又需要用到 Zone0 Page0, 那么还得再次把它换回来。如果一个内存页总是在 Cache 里被不断地刷进来又刷出去, 命中率自然就会非常低。

第二种叫作全联映射, 内存页可以映射到 Cache 中的任何一行上。这种映射关系可以被理解为坐地铁。地铁里的座位是不固定的, 乘客上车后想坐在哪里都行。只要车里还有座位, 就不会有人和你争抢座位。这样一来, 内存页得以长时间地保存在 Cache 中, 命中率就会大大提升。由于全联映射是多对多的关系, 因此它的系统开销非常大。

第三种叫作组联映射, 这是一种折中的方式。组联映射将 Cache 拆分成  $N$  组, 这个  $N$  的取值必须是 2 的  $n$  次方。假设  $N=2$ , 这个映射关系就叫 2-Way Associative。内存页在映射时, 只能选择两组之间的其中一个, 不过一旦选定了一个组, 在组内则是以全联的形式映射的。

举一个不太恰当的例子, 组联映射有点儿像坐火车。你在购票时要先选择座位等级。假设你买了一张二等座的车票, 如果车上空座较多, 又没有其他乘客反对, 你就可以在二等车厢内随便坐。虽然乘坐火车也应该对号入座, 但它不像飞机那么严格, 只要你没坐在一等车厢, 列车员是不会和你较真儿的。当然, 我们不提倡这种乱坐座位的不文明行为, 这里只不过是为例而已。组联映射是在一定范围内, 给予内存选择 Cache 的自由。

#### 14.2.4 CPU 调度算法

进程从大体上可以分成实时进程与非实时进程两类。顾名思义, 实时进程对响应时间的要求很高, 音视频播放就是典型的实时进程的实例。它的调度策略分为 SCHED\_FIFO 和 SCHED\_RR。SCHED\_FIFO 采用了先入先出的机制, 内核按照进程进入队列的顺序执行。SCHED\_FIFO 没有时间片, 如果一个进程被调度器选择执行, 那么它可以运行任意长的时间, 直到执行完毕才会轮到后面的进程, 除非它被更高优先级的进程插队。SCHED\_RR 则是按照优先级排列, 优先级相同的进程以轮询的方式执行。

实时调度适合那些进程数较少, 且延迟影响严重的场景。使用实时调度的最佳实践是

从较低的优先级开始尝试，然后根据实际情况适度地向上提升。切不可直接将进程级别调至最高，以防将系统内部的一些关键性进程阻塞在后面无法运行。

只要系统中有实时进程在运行，它总是优先于非实时进程。所以，在使用实时调度策略执行进程前，需将它们隔离到特定的核心上去，绑定核心时最好从后向前分配，不要让实时调度策略影响到所有的核心，以防止造成进程阻塞。如果有多个进程需要被置成 SCHED\_FIFO，在设置优先级时，要考虑它们之间是否有依赖关系。优先级会影响执行顺序，这一点非常重要。

实时优先级的取值范围是 1~99，按照由低到高的顺序排列。查看和设置实时优先级的方法如下。

```
// 查看进程的实时优先级
# chrt -p <PID>
// 启动进程前设置实时优先级。其中 -f 代表 SCHED_FIFO，-r 代表 SCHED_RR
# chrt -f [1-99] <COMMAND>
# chrt -r [1-99] <COMMAND>
// 调整一个已经运行的进程的实时优先级
# chrt -p -f [1-99] <PID>
# chrt -p -r [1-99] <PID>
```

非实时进程包括 SCHED\_NORMAL、SCHED\_BATCH 和 SCHED\_IDLE。它们使用完全公平调度算法（Completely Fair Scheduler，CFS）来处理。我们依旧可以使用优先级来标识进程的重要程度。其中，我们最为熟悉的 NICE 值，就是此类调度策略的优先级。

SCHED\_NORMAL 用于大多数普通进程。SCHED\_BATCH 则用于执行非交互的批处理任务，它能更好地利用 Cache，而且不会抢占 CFS 中的其他进程。如果你担心影响交互式操作，又不打算降低 NICE 值，那么 SCHED\_BATCH 是一个最好的选择。SCHED\_IDLE 则是所有策略中优先级最低的，这类进程只会在系统负载很低的时候才会执行。

顾名思义，CFS 试图保证每一个进程都能公平地获得相同的时间片。但是这种理念就和优先级发生矛盾了。既然是完全公平，又怎么跳出个优先级来？有优先级还能算是公平吗？

事实上，CFS 维持的是一个时间分配的平均。它认为每个进程都应该获得相同的时间片。但是进程的忙碌程度是不同的，完全平均反倒是不公平了。CFS 采用了虚拟时钟来维持时间量的计算。这个虚拟时钟要慢于真实时钟。它的速度为  $1/N$ ，其中  $N$  是 CFS 挑选的进程数量。假如，CFS 调度了 3 个进程，它们在队列里总共待了 15 秒，换算成虚拟时间就是 5 秒。CFS 根据这个来决定每个进程应当获得多少时间。除此之外，还有一个影响因素就是进程的等待时间。等待时间越长的进程，就越需要被优先考虑。等待时间是存储在红黑树中的。有关红黑树的内容就不在这里展开了。大家可以这样来理解，等待时间最长的进程会被排在红黑树的最左侧，然后依次向右排列。当这个进程被调度并运行了一段时间后，它的等待时间就会变少，此时该进程在红黑树中所处的位置就会被向右移动。



关于优先级的处理方式是这样的：CFS 并不直接使用优先级，而是将其作用于执行时间的衰减系数上。优先级越低，衰减系数越高。也就是说，低优先级的进程在任务执行时，时间消耗得更快。NICE 值的取值范围是 -20~19，按照由高到低的顺序排列。调整 NICE 值的方法如下。

```
// 启动进程前设置 NICE 值
# nice -n [-20-19] <COMMAND>
// 调整一个已经运行的进程的 NICE 值
# renice [-20-19] <PID>
```

## 14.2.5 进程运行在哪个核心上

在调整进程之前，我们需要先了解一下它到底运行在哪个核心上。下面这些方法都是可行的。我们来举一个例子，使用 taskset 指定 yes 运行在核心 31 上。

```
# nohup taskset -c 31 yes > /dev/null &
```

在 top 中，依次按下 Shift+P、F、J 键，选择 Last used cpu 就可以看到进程运行在哪个核心上面，字段名称是 P。其显示效果如下所示。

```
top - 13:20:45 up 3 days, 2:32, 3 users, load average: 1.00, 1.00, 1.00
Tasks: 502 total, 2 running, 500 sleeping, 0 stopped, 0 zombie
Cpu(s): 3.1%us, 0.0%sy, 0.0%ni, 96.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 65920828k total, 65646264k used, 274564k free, 98916k buffers
Swap: 33554424k total, 0k used, 33554424k free, 52482844k cached
```

| PID   | USER | PR | NI | S | %CPU  | %MEM | TIME+     | P  | COMMAND     |
|-------|------|----|----|---|-------|------|-----------|----|-------------|
| 6560  | root | 20 | 0  | R | 100.0 | 0.0  | 162:18.39 | 31 | yes         |
| 29357 | qemu | 20 | 0  | S | 4.0   | 12.5 | 190:17.31 | 15 | qemu-kvm    |
| 7837  | root | 20 | 0  | R | 0.3   | 0.0  | 0:00.05   | 16 | top         |
| 1     | root | 20 | 0  | S | 0.0   | 0.0  | 0:01.60   | 2  | init        |
| 2     | root | 20 | 0  | S | 0.0   | 0.0  | 0:00.01   | 16 | kthreadd    |
| 3     | root | RT | 0  | S | 0.0   | 0.0  | 0:00.55   | 0  | migration/0 |
| 4     | root | 20 | 0  | S | 0.0   | 0.0  | 0:00.26   | 0  | ksoftirqd/0 |
| 5     | root | RT | 0  | S | 0.0   | 0.0  | 0:00.00   | 0  | migration/0 |
| 6     | root | RT | 0  | S | 0.0   | 0.0  | 0:00.18   | 0  | watchdog/0  |
| 7     | root | RT | 0  | S | 0.0   | 0.0  | 0:00.60   | 1  | migration/1 |
| 8     | root | RT | 0  | S | 0.0   | 0.0  | 0:00.00   | 1  | migration/1 |
| 9     | root | 20 | 0  | S | 0.0   | 0.0  | 0:00.29   | 1  | ksoftirqd/1 |
| 10    | root | RT | 0  | S | 0.0   | 0.0  | 0:00.27   | 1  | watchdog/1  |
| 11    | root | RT | 0  | S | 0.0   | 0.0  | 0:00.48   | 2  | migration/2 |
| 12    | root | RT | 0  | S | 0.0   | 0.0  | 0:00.00   | 2  | migration/2 |
| 13    | root | 20 | 0  | S | 0.0   | 0.0  | 0:00.27   | 2  | ksoftirqd/2 |
| 14    | root | RT | 0  | S | 0.0   | 0.0  | 0:00.24   | 2  | watchdog/2  |
| 15    | root | RT | 0  | S | 0.0   | 0.0  | 0:00.48   | 3  | migration/3 |

htop 也可以看。按 Shift+C 组合键，通过方向键和回车键选择 Setup → Columns → Available Columns → PROCESSOR 后，再按 F10 键确认即可，字段名称是 CPU。

如果要单独查看某个进程，只要已知进程名称或者 PID 即可。

```
// 使用 ps 命令的自定义选项 -o 打印进程所占用的处理器
[root@station103 ~]# ps -C yes -o pid,comm,psr
  PID COMMAND          PSR
  6560 yes                31
// 使用 taskset 命令打印进程所占用的处理器
[root@station103 ~]# taskset -pc 6560
pid 6560's current affinity list: 31
```

## 14.2.6 strace 的妙用

前面提到的 strace 除了可以跟踪系统调用外，还是一件“扫雷”利器。有时候，它能帮助你排查一些奇怪的“性能”问题。

下面这个示例是我遇到的一个真实的故障。某天，一位同事向我反映说，他在使用 SaltStack 推送任务时非常卡。在此之前已经排除了几个疑点：Minion 节点的状态是正常的，Master 主机的资源占用率也很低，执行其他系统命令都没问题，只是 SaltStack 有问题。于是，我决定执行 salt --version 来测试。按道理说，这个命令只是查看版本号，应该没有什么特别的影响才对。谁承想，这样一个简单的命令，居然耗费了 8s。下面是我的故障复现。读者可以看到，几乎所有的时间全都消耗在了队列等待上。

```
[root@station103 ~]# time salt --version > /dev/null

real    0m8.231s
user    0m0.186s
sys     0m0.042s
```

于是，我决定使用 strace 命令来跟踪 salt --version 运行的整个过程，并且采用了一个很讨巧的方法，就像下面这样。

```
# strace salt --version
```

这条命令执行完成后，大量的调试信息会直接转储 (dump) 到屏幕上。我并不在意翻滚过去的那些内容是什么，只关心它会在哪里停下来。因为当进程在队列等待时不再会产生任何调用，此时调试信息会停止输出，而我要的就是这个。根据 time 命令的执行结果，我们已知这里会停止 8s 左右，我将复制当前屏幕上的最后几行内容并粘贴到记事本中，然后等待程序继续运行直至完毕。接下来，我将再次执行这个命令。但不同的是，这次我将输出结果保存到文件中。最后，我打开这个结果文件，根据记事本中复制的内容，找到了第一次暂停的位置。以此为断点，在附近进行搜索。

下面这段代码是一个标准的 socket 网络通信连接的过程，我们看 connect() 函数提交了 sin\_port 和 sin\_addr 这两个参数，很显然这是一个 DNS 查询行为。

```
socket(PF_INET, SOCK_DGRAM|SOCK_NONBLOCK, IPPROTO_IP) = 3
```



而 x.x.x.x 和 y.y.y.y 是两个无法访问的 DNS Server，我们再向上翻看，确实有读取 `resolv.conf` 文件内容的操作，如下代码所示。

看来 SaltStack 在执行之前，都会先取得主机信息并尝试名称解析，因为 `/etc/resolv.conf` 文件中配置了不可访问的 DNS Server，叠加上 DNS 的超时时间和重试次数，等待这么久也不足为奇。删除了 `/etc/resolv.conf` 文件中的错误条目后，故障便立即解除了。

这个解决问题的思路很巧妙。首先利用 `time` 观察到了时间瓶颈。既然时间消耗在了队列等待上，就说明这段时间没有任何交互，那么 `strace` 的调试信息就会停止输出，相当于程序自己设置了一个断点。第一次执行的目的在于找到这个断点的位置，第二次则是为了获取完整的调试信息。最后，根据断点位置在其附近查找隐藏着的、真正的故障原因。

### 14.3 内存

基本上，应用程序极少会将数据直接写入存储。和 Cache 的作用一样，内存是连接计



算节点（处理器）和存储节点（磁盘）的关键一环。如何充分地利用好内存资源，对于系统整体性能的影响是非常大的。所以在性能校准的工作中，我们对内存的关注力度一点也不会比 CPU 少。

### 14.3.1 NUMA

NUMA 技术可能是一个很容易引发讨论的话题。有些人觉得 NUMA 很好，而对于一些 MySQL 和大数据的管理员来说，对 NUMA 的抱怨往往会更多。

为什么要使用 NUMA？这还得从 SMP（Symmetric Multi-Processor，对称多处理器）说起。要评价一台服务器的性能优劣，很多人第一时间就会想到 CPU 的工作主频，也就是它的时钟频率。因为主频越高，它的时钟周期就越短。假如 CPU 的主频是 100MHz，一个时钟周期就是 10ns。如果主频提升到 1GHz，对应的时钟周期就会缩短到 1ns。我们知道计算机指令是以时钟周期为单位计算的，所以时钟周期越短，程序运行得就越快。但是由于受到制作工艺的限制，主频不能无限度地提升。为了突破瓶颈，人们开始采用多处理器架构来解决问题。

SMP 是一种典型的多处理器架构，每个处理器都是对等的，并通过总线连接来共享物理内存，因此 SMP 也被称为 UMA（Uniform Memory Access，一致性内存访问）。由于 SMP 的总线和内存均为共享，这里会成为性能瓶颈，使得 CPU 的扩展能力受到限制。

如果我们把 CPU 当作人，把内存看作火锅，那么，SMP 就是大家共用一个涮锅。共享资源对人数是有限制的。锅就这么大，如果人数过多就会产生争用的现象。NUMA 相当于给每个人分配一个小锅，一人一锅形成一个 Node。吃自己锅里的（Local Access）要比吃别人锅里的（Remote Access）方便，NUMA 使用 Distances 这个概念来描述 Local Access 和 Remote Access 之间的差别。我们可以使用命令 `numactl --hardware` 查看各 Node 之间 Distances 的情况。

```
[root@station103 ~]# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
node 0 size: 32722 MB
node 0 free: 3985 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
node 1 size: 32768 MB
node 1 free: 27361 MB
node distances:
node    0    1
  0:   10   20
  1:   20   10
```

`numactl` 的策略分为两大类。第一类是绑定 CPU 的策略。`cpunodebind` 指定进程只运行在某个 Node 上面，`physcpubind` 指定进程与运行在哪个核心上。第二类是绑定内存的策

略。interleave 会将各个 Node 上的内存轮询分配给进程使用，使用 interleave 需要指定在哪些 Node 上轮询。

localalloc 则是指进程只能使用本地内存，这也是默认策略。开启了 NUMA 的系统在启动时会先使用 interleave 以防止启动失败，启动成功后会恢复成 localalloc。preferred 则是优先使用指定的 Node，当内存不够时再去分配其他 Node 上的内存。mbind 用来指定进程从哪一个节点上分配内存，通常它会和 cpunodebind 配合使用。

Linux 默认的 NUMA 内存分配策略是 localalloc，也就是说大家各吃各的，文明用餐，谁也不碍谁的事儿。不过，这个看似很正常的分配策略却给 MySQL 和大数据带来了麻烦。因为小锅没有大锅盛得多，如果一个食量很大的进程，对内存的访问需求超过了本地的内存总量，而又它不能吃别人锅里的，怎么办？这里有两种办法。第一，把自己锅里的菜捞出来再放新的。但是如果菜没煮熟，就算你现在捞出来，一会儿还得再放进去。第二，再找一个叫作 SWAP 的锅，这个锅的火力没有原来的大，涮起菜来会更慢。不管怎样，现在系统提供的锅对于这个进程来说都太小了。两种方式在内存中的表现就是数据被频繁地换入换出。要么使用速度更慢的 SWAP，要么 Kill 掉部分进程以释放内容，如果做了这么多还不够，那你就等着应用挂掉吧。

numastat 命令用来查看当前 NUMA 命中率的情况，你可以根据实际情况来调整现有 NUMA 策略。自定义 NUMA 策略可参看 NUMACTL (8) 的手册。

通过下面这个示例我们可以看到，Node 1 的命中率明显很差，相比 Node 0 差出两个数量级。numa\_foreign 表示来自另外一个 Node 的内存申请。每出现一个 numa\_foreign，在其他 Node 上就必定会有一个 numa\_miss。在两个 Node 的场景中，Node 0 的 numa\_foreign 等于 Node 1 的 numa\_miss。Node 1 上 other\_node 的占比已经达到了内存访问总量的三分之一，这说明 Node 1 上对应的程序比较吃内存。可通过前面介绍的方法，查看到底是运行在 Node 1 上的哪些程序导致的，然后根据实际情况进行对应的调整。另外要说明的一点是，默认策略开机时会先启用 interleave，所以 other\_node 这个值要持续观察，根据变化才能确认结论。

```
[root@station103 ~]# numastat
```

|                | node0     | node1     |
|----------------|-----------|-----------|
| numa_hit       | 248985970 | 445180828 |
| numa_miss      | 2198273   | 219593575 |
| numa_foreign   | 219593575 | 2198273   |
| interleave_hit | 20626     | 20676     |
| local_node     | 248975577 | 445164869 |
| other_node     | 2208666   | 219609534 |

默认的 NUMA 策略对于多实例的应用场景是最为有利的（典型的的就是 KVM）。对于那些大型的单实例应用来说，依旧使用默认策略，至少会浪费掉一半的内存资源，还可能引发资源不足。虽然有些管理员采用了 interleave，但这未必是最优模式，我个人认为任务拆分还是最好的解决途径。

### 14.3.2 Cache 和 Buffer

`free` 是我们最常用的一条命令，大家都知道它是用来查看剩余内存的。不过 `free` 的输出中有两个特别的字段——Cache 和 Buffer。不少朋友把它们统称为缓存。但它们之间到底有什么差别呢？

```
[root@station101 ~]# free
```

|                    | total    | used   | free     | shared | buffers | cached |
|--------------------|----------|--------|----------|--------|---------|--------|
| Mem:               | 65920828 | 870596 | 65050232 | 0      | 71960   | 256548 |
| -/+ buffers/cache: |          | 542088 | 65378740 |        |         |        |
| Swap:              | 33554424 | 0      | 33554424 |        |         |        |

Cache 的英文原意是贮藏的地方。说到贮藏，我就会不由自主地想到地窖。地窖里面一般储备的是食品、饮用水或者金银细软之类的财物以备不时之需，当战争或者灾难突然降临的时候，人们就可以从地窖里快速获取物资。Buffer 的英文原意是缓冲。例如，电梯间的底部都有一个缓冲器，当电梯失控以自由落体的速度下落时，它能起到减缓伤害的作用。

虽然 Cache 和 Buffer 在计算机领域都被统称为“缓存”，但是从英文原意上我们还是可以看出两者之间的区别的。Cache 是为了更快地获取数据，它通过缩短调用时间来获得性能的提升。比如说日常生活中，很多人喜欢把手机揣在衣兜里。这时的衣兜就是一个 Cache，因为手机会随时拿出来用，如果把它放到背包里，取用时就显得很麻烦。而 Buffer 则是主要用于数据的临时存储，通过合并操作以减少不必要的 I/O 来提升性能。Buffer 可以被看作网购里的购物车，当用户选择购买一件商品时并不会立即支付，而是先将其放到购物车里，等所有商品都挑选完毕后，再统一从购物车下单。所以，Cache 的本质是快取，它的最终流向是数据调用。Buffer 的本质是暂存，它的最终流向是写入存储。

另外我们还要避免陷入两个误区。第一个误区——Cache 就是读，Buffer 就是写。Buffer 通常用于写入，但 Cache 却是可读可写的。例如，CPU 的 Cache 你能说它是只读的么？第二个误区——Cache 和 Buffer 是互斥的。其实，站在不同的角度上看，这两个概念是可以相互转换的。比方说，阵列卡的 Cache 对于存储的写入来说不就是 Buffer 么？再举一个典型的例子，就是以前我们常用的刻录机。购买刻录机有一个特别重要的指标——就是看它的 Cache 有多大。因为根据桔皮书的规范，光盘中扇区空隙不能大于 100 $\mu$ m。刻录时如果不能连续从磁盘中读取数据，空白时间过长，光盘就会报废，俗称“飞盘”。为了避免这个问题，需要先将数据放到“缓存”上，读“缓存”要比读磁盘快得多，这样就可以抑制“飞盘”的产生。对于刻录机来说，这个“缓存”应该叫 Cache。而对刻录软件来讲，它又是 Buffer。如果因“缓存”不足而出现“飞盘”，刻录软件一定会返回 Buffer under run error 这个错误。因为 Cache 里的数据最终还是要写到光盘里的，所以程序在这里才会使用 Buffer 这个词。

为了避免误区，最好的方法还是要理解它们的原意。作为“缓存”，它们都提供了相对



高速的存储空间。但从硬件的角度上来说，所有的“缓存”都应该叫作 Cache。不论是读还是写，Cache 作为硬件本来就是为了提速的，它并不参与 I/O 合并等一系列操作。而在软件层面上，用于数据读取的应归属于 Cache，负责 I/O 合并、将数据写入存储设备的则是 Buffer。

### 14.3.3 虚拟地址空间

应用程序运行前要先申请一定的内存空间。这和申请电子邮箱一样，每个进程都希望能够使用独立的地址空间，而不是和其他进程共享。因此，如果直接使用物理内存的话，进程相互之间是可见的。想想私人邮件被放到公共邮箱中是一件多么可怕的事情。另外，直接使用物理地址空间，会出现不可调和的地址冲突现象。这就要求程序员必须先同其他进程协商内存占用，然后才能去申请地址。比如，你要给一个变量赋值之前，必须知道该变量指向的地址空间是否正在被其他进程所使用。如果写一个应用程序还要考虑这种系统逻辑，那么开发门槛就太高了。

为此，系统提供了虚拟地址空间来解决上述问题。进程相互之间是隔离的，彼此感知不到对方的存在。程序员在编写代码时，无须考虑其他进程对物理内存的占用，也不用关心物理内存是如何管理的。对于每一个进程来说，整个地址空间都是它的。这个空间的大小和字长有关。32 位系统的地址空间是  $2^{32}$ B 即 4GiB，其中内核地址空间要占用 1GiB，应用程序所能使用的空间上限就是 3GiB。这也就是为什么在 32 位系统下最大可用内存无法大于 3GiB 的原因了。这个空间对于类似 Oracle 这样的进程来说实在是太小了。64 位系统则能够提供足够大的地址空间，来满足大型应用程序对内存的需求。

既然是虚拟地址，那么最终它还是要和物理地址之间建立映射关系。存储这种关系的数据结构称为页表。两种地址空间都被内核分成多个小块，作为组成空间的最小单位。不过，它们在称呼上有所区别。虚拟地址的内存块叫作分页（Page），物理地址的内存块则被称作页帧（Page Frame）。

注意：虚拟地址空间远远大于系统实际拥有的物理内存的大小。以 32 位系统为例，分页的大小为 4KiB，一个进程 4GiB 的虚拟地址空间就需要一百万条存储记录。每个进程都有自己的页表，物理内存光是保存页表就不够用了。事实上，就像电子邮箱的空间一样，进程根本就用不了多少分页。所以，绝大多数分页都没有关联到页帧上去。而且，分页和页帧之间不是直接对应的。否则，那么多关联关系，管理起来是十分困难的。系统采用了多级页表的管理方法，就像下面这样。

PGD -> PMD -> PTE + OFFSET

PGD（Page Global Directory）被称作全局分页目录。每个进程都会有且只有一个 PGD，它并不存储实际物理地址，而是作为一个指针指向它的下级数据结构——PMG（Page

Middle Directory)，也就是中间分页目录。和 PGD 一样，PMG 也是指向它的下级数据结构的，对于多级页表结构有可能会有多个 PMG，直到指向最终的存储地址的 PTE (Page Table Entry)。OFFSET (偏移量) 则是用来标记页帧中的某个位置，内存寻址就是依靠地址加偏移量来定位的。

#### 14.3.4 大页

多级页表访问的缺点也是明显的，那就是速度太慢。每次访问内存总是要读取多个页表。为了解决这个问题，CPU 要把那些频繁使用的映射关系进行缓存。MMU (Memory Management Unit, 内存管理单元) 负责地址关系映射的工作，TLB (Translation Lookaside Buffer, 转译后援缓冲器) 负责缓存这些映射关系。不过 TLB 不够大，如果内存容量很大，TLB 根本存储不了太多的条目。一旦超出了 TLB 的存储极限，就会发生 TLB miss。也就是说，CPU 只好去访问内存上的页表。频繁的 TLB miss 会使程序性能迅速下降。为了让 TLB 能够存储更多的条目，可以加大单个内存分页的大小，也就是我们所说的大页。

64 位系统支持分页大小有 4KiB、8KiB、64KiB、256KiB、1MiB、4MiB、16MiB 和 256MiB。/proc/meminfo 文件中实时反映了大页的相关信息。要支持大页的使用，用户可以采取两种系统调用方法。第一种是共享内存系统调用，也就是 shmat() 和 shamget()。另外一种就是 mmap()。如果是采取第二种方法，SE 则需要使用如下命令挂载一个伪文件系统来进行支持。

```
# mount -t hugetlbfs [-o fs_options] none <MOUNT_POINT>
```

使用大页后，内核会提供一个 Huge Page Pool 供应用程序申请。这个 Pool 是由若干 Huge Page 组成的，需要事先声明 Huge Page 的数量。我们可在 /etc/sysctl.conf 文件中通过定义 vm.nr\_hugepages 来实现。

#### 14.3.5 内存分配

内核在分配内存时，必须了解页帧的使用情况。Free 表示该页帧是空闲的，可以立刻分配给申请者。Inactive Clean 和 Inactive Dirty 都表示该页帧已经被使用过了。不同的是，Clean 代表页帧里的数据和磁盘是一致的，也可以当作空闲页被分配出去，而 Dirty 则不行。Active 则代表该页帧正在使用中，也是无法释放的。

除了要了解页帧的使用状态以外，内核还要尽快完成分配工作。显然，分配连续的页帧是最好的方式。如果应用程序要申请 8 个页帧，内核要在连续空闲页帧大于 8 的区域调取才是最快的。这就产生了一个问题，我们知道磁盘会产生碎片，内存也一样，那么内核如何才能快速找到连续空闲页帧的区域呢？

Linux 使用了 Buddy System 来完成这个任务。页帧总是被成对分配的，成对的页帧称

为 Buddy。分配页帧时，Buddy System 总是将它们从更大的页帧上成对地拆分。而在页帧回收时，Buddy System 会检视它们和已有的空闲页帧是否存在 Buddy 关系，以便将它们重新合并成一个更大页帧，以此来防止内存碎片的产生。

在分配方式上，Buddy System 和网络地址规划很像。假设我们有一段连续的空闲页帧 PF 0~7。这时，应用 A 向系统申请了两个页帧，Buddy System 会将 PF 0~7 分成两半——PF 0~3 和 PF 4~7，再将 PF 0~3 对半拆开，把 PF 0~1 分给应用。不久，应用 B 向系统申请了四个页帧，Buddy System 则会将 PF 4~7 分配出去，而不是 PF 2~5。因为 PF 2~5 不能组成一个 Buddy。当应用 A 结束并返还 PF 0~1 时，Buddy System 会将其和空闲的 PF 2~3 重新合并成 PF 0~3。

既然进程访问的是虚拟地址空间，如果它访问的分页不在物理内存中，此时系统会产生一个主要页错误 (Major Page Fault)。这时要触发 I/O 操作，系统会从磁盘读取相关数据，然后将其载入内存并建立物理内存地址与虚拟地址的映射关系。注意：不只是磁盘，从 SWAP 向内存载入时也会产生主要页错误。如果分页已在物理内存中，只是没有建立映射关系，系统会产生一个次要页错误 (Minor Page Fault)。通常父进程 fork 多个子进程时，在只读模式下，它们共享同一个内存中的数据时，可能由于某些子进程还未建立起映射关系，就会出现次要页错误。

通过 sar 命令可以查看分页错误的情况，majflt 代表主要页错误，pgpgin 和 pgpgout 反映了内存和磁盘交换的情况。如果你在同一时刻发现它们三个的数值都很高，说明系统在内存使用上出现了问题。另外，如果执行 vmstat 观察，发现 si 和 so 的数值都很高，说明 SWAP 的置换很多。很明显，这也是内存不足引起的。

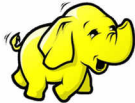
// 通过 sar 命令可以查看分页错误的情况

```
[root@station101 ~]# sar -B -s 16:00:00 -e 17:00:00
```

```
Linux 2.6.32-431.el6.x86_64 (station101.example.com) 08/09/2017 _x86_64_
(32 CPU)
```

|             | pgpgin/s    | pgpgout/s | fault/s | majflt/s | pgfree/s | pgscank/s | pgscand/s |
|-------------|-------------|-----------|---------|----------|----------|-----------|-----------|
| 04:00:01 PM | pgpgin/s    | pgpgout/s | fault/s | majflt/s | pgfree/s | pgscank/s | pgscand/s |
|             | s pgsteal/s | %vmeff    |         |          |          |           |           |
| 04:10:01 PM | 0.00        | 0.00      | 0.53    | 12.82    | 0.00     | 7.28      | 0.00      |
|             | 0.00        | 0.00      |         |          |          |           |           |
| 04:20:01 PM | 0.00        | 0.00      | 0.45    | 6.39     | 0.00     | 5.38      | 0.00      |
|             | 0.00        | 0.00      |         |          |          |           |           |
| 04:30:01 PM | 0.00        | 0.00      | 0.41    | 6.39     | 0.00     | 5.38      | 0.00      |
|             | 0.00        | 0.00      |         |          |          |           |           |
| 04:40:01 PM | 2.61        | 1.68      | 46.31   | 0.10     | 20.18    | 0.00      |           |
|             | 0.00        | 0.00      |         |          |          |           |           |
| 04:50:01 PM | 1088.37     | 2625.83   | 1569.89 | 0.65     | 12005.61 | 0.00      |           |
|             | 0.00        | 0.00      |         |          |          |           |           |
| Average:    | 218.15      | 525.67    | 328.29  | 0.15     | 2408.25  | 0.00      |           |
|             | 0.00        | 0.00      |         |          |          |           |           |





### 14.3.6 内存回收

解决内存不足最偷懒的办法就是增加内存。当然，这不是我们本节的重点。理解了内存回收的工作机制后，我们可以通过调整内核参数来解决一些问题。前面我们提到内存分配时要尊重页帧的使用情况。由于 Inactive Dirty 脏页上的数据和磁盘不一致，导致可用页帧数量的减少。一个大的写操作会产生大量的脏页数据，如果它们能及时回收就好了。新版内核中使用 per-BDI flush 线程来处理脏页数据的写回。其中，我们可以通过修改以下这些参数来控制 per-BDI flush 线程的写回行为。

#### (1) vm.dirty\_writeback\_centisecs

内核会定期召唤 per-BDI flush 线程执行回写任务，该参数用来表示这个周期是多久。单位是百分之一秒，默认值为 500，即 5 秒。如果系统是持续写入且监控数据生成的图形呈锯齿状，适度降低该数值可擦除尖峰，让曲线更加平滑。

```
[root@station101 ~]# cat /proc/sys/vm/dirty_writeback_centisecs
500
```

#### (2) vm.dirty\_expire\_centisecs

per-BDI flush 线程并非只要一醒过来，见着脏页就回写。脏数据要待够一定的时间才有资格被回写。该参数就是用来控制这个时长的，默认是 3000，也就是 30 秒。

对于写入负载很重的应用来说，适度减小是有益的，但同时也会引发 I/O 开销。如果系统内存较大、非连续写入并写入数据量不大，则可以适度提高一些。总之，切记不要调过头。

```
[root@station101 ~]# cat /proc/sys/vm/dirty_expire_centisecs
3000
```

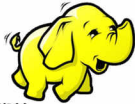
#### (3) vm.dirty\_background\_ratio & vm.dirty\_ratio

这两个参数表示当脏数据在系统内存中的占比达到百分之多少时开始回写磁盘，只是两种回写模式有所不同。

根据 Kernel-doc 的官方解释，vm.dirty\_ratio 指在一个强制回写发生之前所能允许的脏页占比是多少。vm.dirty\_background\_ratio 是指当脏页占比因超过 vm.dirty\_ratio 而触发强制写回后所能允许的脏页占比是多少。前者是让进程自己进行一个强制回写操作，后者则是调用 per-BDI flush 在后台写入。两者的默认值分别是 20 和 10。

```
[root@station101 ~]# cat /proc/sys/vm/dirty_ratio
20
[root@station101 ~]# cat /proc/sys/vm/dirty_background_ratio
10
```

这两句话怎么去理解呢？假设我有 64GiB 内存，当脏页使用到 6.4GiB 时，内核就开始回写了。这时候，你内核回写你的，我脏页增加我的，谁也碍不着谁。如果脏页增加的速



度大于回写，那么脏页会越来越多。如果达到 12.8GiB 时，内核就会说：“这样下去不行，你脏页先不要增加了，等我先把现有的脏页处理完，你再增加也不迟。”这时候就会产生一个阻塞，如果现有脏页没有提交到磁盘，就不允许再增加新的脏页了。

一方面，回写肯定会增加 I/O 的开销，`vm.dirty_background_ratio` 的值不应设置得过小，否则会频繁增加回写次数。另一方面，如果 `vm.dirty_ratio` 的值设定得很大，一旦触发了强制回写会导致非常高的延时，因为要回写的数据量太多了。所以，调整的时候一定要注意这一点。对于具有 NVRAM 或 SSD 的应用场景，可以适度降低这两个数值，从而有效降低高延迟。以下设置仅供参考。

```
vm.dirty_writeback_centisecs = 5
vm.dirty_expire_centisecs = 20
vm.dirty_background_ratio = 5
vm.dirty_ratio = 10
```

对于那些日常不频繁、但一波写入量很大且为慢速存储的批处理任务，可以适度调高 `vm.dirty_ratio` 的值。以下设置仅供参考。

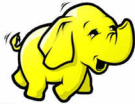
```
vm.dirty_writeback_centisecs = 10
vm.dirty_expire_centisecs = 30
vm.dirty_background_ratio = 5
vm.dirty_ratio = 60
```

### 14.3.7 内存超配了怎么办

鲁迅笔下的孔乙己被邻居的孩子们围住时，便会给他们一人一颗茴香豆。孩子们吃完豆，仍然不散，眼睛都望着碟子。这时的他却着了慌，伸开五指将碟子罩住并言道：“不多不多！多乎哉？不多也。”

事实上，系统内存也有被应用抢光的危险。内核根据 `overcommit_memory` 决定内存分配的策略，策略总共有三个。0 表示内核有限度的允许内存超配。它会将当前空闲、预留以及可以释放的内存全部加起来，如果能够满足超配就分，如果不行就返回失败，属于“勒紧了裤腰带过日子”的类型，这也是内核默认的分配策略。1 表示内核假充大款，无限度、无原则地允许内存超配申请。不论何时，内核永远不会对应用说 NO，直到内存用光自己装不下去了为止。这个值的适用场景是科学计算。如果内存分配错误，计算就会终止并重新返工，前面付出的时间也就都白费了。设置成策略 1 后，内核会尽最大力量提供分配保证，也算得上是“鞠躬尽瘁、死而后已”了。2 则表示内核永远不允许内存超配，它的分配底线是 SWAP 加上物理内存的百分比，百分比由 `overcommit_ratio` 控制，默认是 50%。

```
[root@station101 ~]# cat /proc/sys/vm/overcommit_memory
0
[root@station101 ~]# cat /proc/sys/vm/overcommit_ratio
50
```



### 14.3.8 为什么会产生 OOM

如果内存被分配光了，就会产生 OOM。

OOM 全称是 Out of Memory。导致 OOM 的原因有两种：一种可能是内存太少，原本就不够用；另一种则是内存泄漏——也就是应用程序申请的内存存在用完后一直不释放造成的。内存泄漏随后会引发内存溢出，即申请的内存超越了系统所能提供的范围。一种比较常见的溢出错误是 Segment Fault，它表示进程需要访问的内存地址不在它的虚拟地址空间范围内，属于越界访问。

不同的语言对于内存回收的处理机制不同，有些内存泄漏是因为程序员忘记了回收，而有些则是因逻辑问题导致的（比如说我释放了一个内存块，然后又继续引用其中的内容）。可以使用工具 valgrind 来测试一个程序是否存在内存泄漏或者内存溢出的问题。

```
# valgrind --tool=memcheck --leak-check=full <Program>
```

memcheck 工具会跟踪所有 Heap Block 对 malloc() 或者 new() 调用的响应。当程序退出后，它就会知道哪些 Block 没有被释放。命令执行后，你可能会看到类似如下这样的结果。

```
LEAK SUMMARY:
    definitely lost: 48 bytes in 3 blocks.
    indirectly lost: 32 bytes in 2 blocks.
    possibly lost: 96 bytes in 6 blocks.
    still reachable: 64 bytes in 4 blocks.
```

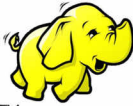
其中，definitely lost 是说这些 Block 找不到指向它们的指针。指针是用来存储内存地址的，如果一个 Block 没有指针指向，意味着它们现在被丢弃了，没有人去回收它们。出现这种问题，应当由程序员负责处理。indirectly lost 的出现并非是这些 Block 没有指向它们的指针，而是它们上一级的指针丢失了。例如，一个二叉树丢失了根节点，虽然子节点还有指针指向，但实质上这些子节点也等同于间接丢失。剩下两个事件类似于 Warning 级错误，如果出现也应当引起程序员的注意。

当 OOM 出现以后，内核又是怎么处理的呢？为了保证系统正常运行，内核会调用 oom\_killer() 来杀死某一个进程，借此腾出空间来，以防止系统出现 Kernel Panic 的现象。这个策略由 panic\_on\_oom 决定。它的默认值为 0，代表当出现 OOM 时调用 oom\_killer()。如果设置成 1，当 OOM 出现时就会触发 Kernel Panic，但不包括因 mempolicy 或 cpusets 限制产生的 OOM。如果设置成 2，则代表任何情况的 OOM 都会导致 Kernel Panic 的发生。

```
[root@station101 ~]# cat /proc/sys/vm/panic_on_oom
0
```

如果要调用 oom\_killer() 执行屠杀任务，究竟干掉哪个进程好呢？总不能击鼓传花或是点到谁就算谁吧？





Kernel 使用 `oom_score` 来决定哪一个进程是最终的那个倒霉蛋。分数最高的进程会被最先 kill 掉。计算 `oom_score` 是一套复杂的逻辑，它和进程的运行时间、NICE 值以及子进程的数量等一系列因素都有关系。当然，我们也可以人为控制其中一部分因素。2.6.36 之前的内核版本都是通过调整 `oom_adj` 来影响 `oom_score` 的。它的值定义在 `-17~15` 之间，`-17` 代表该进程禁用 OOM Killer。之后的新版本，内核将参考 `oom_score_adj` 来进行决策。即便修改 `oom_adj`，最后也会转换成相应 `oom_score_adj` 的值。我们再次着重重申，`oom_score` 的计算是基于多个条件的。关于 `oom_score` 等于 `2oom_adj` 的说法，要注意其所指的那个 `oom_score` 只是一个加权值，并非是最终的 `oom_score`。

这些相关文件都位于 `proc` 目录的每个进程的子目录下面。

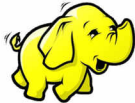
```
[root@station103 ~]# ll /proc/1/oom_*
-rw-r--r-- 1 root root 0 Jul 27 09:43 /proc/1/oom_adj
-r--r--r-- 1 root root 0 Jul 27 09:43 /proc/1/oom_score
-rw-r--r-- 1 root root 0 Jul 27 09:43 /proc/1/oom_score_adj
```

下面这段代码可以告诉你最先被 kill 掉的 TOP 10 进程是谁。

```
#!/bin/bash
echo PID SCORE ADJ SCORE_ADJ NAME
for i in `ls /proc/ | grep -P "[0-9]"`
do
    pid=`echo $i`
    name=`ps -p $i | awk -v var=$i '{ $0 ~ var { print $NF } }`
    score=`cat /proc/$i/oom_score`
    adj=`cat /proc/$i/oom_adj`
    score_adj=`cat /proc/$i/oom_score_adj`
    if [ $? -eq 0 ]; then
        echo $pid $score $adj $score_adj $name
    fi
done 2> /dev/null | sort -nr -k2 | head -10
```

执行之后，我们可以看到类似如下这样的输出结果。只要稍作调整，我们也能知道哪些进程最不容易被干掉。如果你不希望自己的进程在 OOM 出现时被内核“招待”，可使用 `echo` 命令临时调整 `oom_score_adj`，将分值降低。

```
[root@station103 ~]# sh oom_score.sh
PID SCORE ADJ SCORE_ADJ NAME
29357 82 0 0 qemu-kvm
27203 8 0 0 qemu-kvm
27268 7 0 0 qemu-kvm
27235 7 0 0 qemu-kvm
12953 5 0 0 bundle
13167 4 0 0 bundle
13164 4 0 0 bundle
13161 4 0 0 bundle
13155 4 0 0 bundle
13152 4 0 0 bundle
```



## 14.4 存储

不管怎样，最终的数据还是要落到存储上。但是相比起 CPU 和内存来说，这里的读写速度就更慢了。一方面，我们要保证数据存储的完整性，也就是不能丢数据。另一方面，我们又希望数据能尽快地完成读写操作，最大化吞吐的同时，还要尽可能地减少延时时间。看来，我们要做的工作还不少呢。本节内容涉及几种常见的磁盘及 I/O 调度算法、日志模式，以及其他一些和队列深度相关的系统内核参数的调整。下面，我就为你一一道来。

### 14.4.1 磁盘调度算法

传统机械盘是通过磁头来完成数据读写的。读写数据前，要先将磁头移到对应的磁道上，这个过程就是所谓的“寻道”。寻道方式对磁盘读写性能的影响很大，如何控制寻道逻辑有很多不同的调度算法，常见的调度算法有以下四种。

#### 1. FCFS

FCFS 全称为 First Come First Service，即先到先得，哪里先发出请求就先响应哪里。FCFS 在时间响应上保证了公平性，但磁头会因此不断地反复移动，平均寻道距离大，且服务时间偏长。该算法只适用于顺序读写且磁盘碎片较少的场景。

#### 2. SSTF

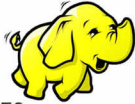
SSTF 全称为 Shortest Seek Time First，即最短寻道时间优先。顾名思义，它永远优先响应距离自己最近的 I/O 请求。SSTF 寻道时间最短，但是无法保证响应时间，如果一个距离磁头较远的请求 A 在发出之后，磁头附近紧跟着又出现了很多新的请求，会导致请求 A 很长时间得不到响应。这样就会产生 I/O “饥饿现象”。

#### 3. SCAN

SCAN 也被称为电梯调度算法，磁头就像电梯一样，从里向外，然后再从外向里，如此往复运动。它总是先响应和自己运动同向且距离最近的 I/O 请求。SCAN 的寻道性能比较好，可有效避免“饥饿现象”的出现，但是如果请求总是出现在远离磁头的一端则对 SCAN 十分不利。

#### 4. C-SCAN

C-SCAN 是对 SCAN 的一种改进。假设 I/O 请求在磁道上的分布是均匀的，当磁头从一端移动到另一端后，此时如果立即反向运动就显得有些不太合理。比方说，磁头从盘面的外圈向里移动，靠里面的数据刚刚才处理过，短时间内不大可能再次出现 I/O 请求，而外圈却已经积累了很多的 I/O 请求。那么，再从里向外去寻道就不科学了。C-SCAN 规定磁头始终是单向移动，当磁头到达终点后会返回起点，开始新一轮的扫描。C-SCAN 消除了磁道两端的不公平性，它适用于随机写这类 I/O 分布相对均匀的场景。



## 14.4.2 I/O 调度算法

I/O 调度算法要做好两件事情，第一要尽可能地快速响应，第二不能让某一个进程等太久。CentOS 6 中支持的 I/O 调度算法共有四种，它们是基于磁盘子系统的。不同的应用场景使用不同的调度算法，调整算法可以使用 `echo` 命令完成。调度算法的参数位于 `iosched` 子目录中。

以下命令是将调度算法从 CFQ 修改成 Deadline 的过程。

```
[root@station101 ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
[root@station101 ~]# echo deadline > /sys/block/sda/queue/scheduler
[root@station101 ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory [deadline] cfq
```

### 1. CFQ

CFQ (Completely Fair Queuing) 被称作完全公平列队，它提供了三种不同等级的调度策略，优先级从高到低分别为 Real-time (RT)、Best-effort (BE) 和 idle。只有当高等级的 I/O 操作执行完毕后，才可以执行低等级的 I/O 操作。CFQ 为每一个执行 I/O 操作的进程分配一个时间片，在这段时间内，默认进程每次最多可以提交八个请求。CFQ 为了保证公平，不管是否存在都会等待 I/O 请求。这时即便其他进程发出 I/O 请求处理，它也不会理会。

在大部分场景下，CFQ 的表现可以算得上中规中矩。它试图将 I/O 带宽均匀地分配到每个进程上，所以很适合离散读的应用。但是它太过于追求公平，所以对延迟的控制力不强。不太适用于数据库、SSD、存储设备以及虚拟机这些应用场景。

CFQ 中有如下几个重要的参数。

#### (1) back\_seek\_max

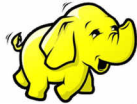
前面我们已经讲过，反向查询通常对性能有负面影响，因为它比正向查询的重置时间要长很多。但如果负载较小，则 CFQ 仍执行此查询。该参数用来控制允许反向查询的距离。单位是 KiB，默认为 16KiB。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/back_seek_max
16384
```

#### (2) back\_seek\_penalty

由于反向查询的效率低，所以为了尽可能不产生反向查询，CFQ 提供了这样一个惩罚值。它是一个乘数（默认为 2），如果磁盘是依据 I/O 请求的距离远近来决定移动走向（例如使用 SSFT 调度算法），向后移动的距离要先乘上这个惩罚值，才能和向前移动的距离进行比较。假设磁头前后各有一个 I/O 请求，向前移动是 10，向后移动是 6，因为  $10 < 6 \times 2$ ，所以磁头最终还是向前移动。





```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/back_seek_penalty
2
```

### (3) fifo\_expire\_async & fifo\_expire\_sync

这两个参数用来控制 I/O 请求的等待时间。当发起一个 I/O 请求时，调度器会开始计时，如果到了过期时间还没有被响应，调度器会将该请求移动到调度表中，以防止进程饥饿。两者的单位都是毫秒，默认值分别为 250 和 125。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/fifo_expire_async
250
[root@station101 ~]# cat /sys/block/sda/queue/iosched/fifo_expire_sync
125
```

### (4) quantum

quantum 用来控制设备的队列深度，默认是 8。适度增加队列深度对基于 SSD 的随机写是有帮助的。但这样做对顺序写的应用场景来说会增加延迟。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/quantum
8
```

## 2. Deadline

Deadline 调度算法会将读写两种 I/O 请求区分开，尽可能地批量化处理某一类请求，并保证在一定延迟内切换，不会让任何一个请求因为等太久而被饿死。注意：只有当请求进入 I/O 调度程序后，延迟才会开始计算。

由于大多数程序阻塞的原因是在等待数据，所以 Deadline 读取的优先级默认要高于写入。它会按批次处理一批连续读或写的 I/O，并按 LBA（逻辑区块地址）的顺序单向递增处理。完成一批后，调度器会检查是否会有等待已久的写请求，如果有就会先执行这些写请求，防止它们被饿死。虽然读取的优先级高于写入，但如果执行写入的同时又产生了读请求，读取操作也要等到这批写入操作完成后才可以。

Deadline 在优先读请求的同时又兼顾了写请求的响应需求，所以非常适用于类似数据库这种对响应要求比较高的系统环境。

Deadline 中有如下几个重要的参数。

### (1) fifo\_batch

该参数用于控制一个批量操作任务的读写数。默认为 16。调高该数值在提升吞吐量的同时，也会增加延迟。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/fifo_batch
16
```

### (2) front\_merges

Deadline 可对多个请求进行合并处理，这样可以提升吞吐量，但付出的代价就是必须

花时间去排序。如果你的请求压根儿就不连续，那就没有合并的可能了，直接禁用掉就好了。禁用后会带来随机读写的特性，不过请你一定要了解清楚实际情况后再做出禁用的决定。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/front_merges
1
```

### (3) read\_expire & write\_expire

这两个参数用来控制读写的过期时间，单位都是毫秒。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/read_expire
500
[root@station101 ~]# cat /sys/block/sda/queue/iosched/write_expire
5000
```

### (4) writes\_starved

该参数用来控制读写操作次数的占比，即读几批后再执行一次写。这个值越大越倾向于读操作。不过，要注意的是调太大也没有什么用。如果 write\_expire 超时，写操作还是会被调度的。

```
[root@station101 ~]# cat /sys/block/sda/queue/iosched/writes_starved
2
```

## 3. NOOP

NOOP 算法对 I/O 几乎不做任何干预，只是在 FIFO 队列的基础上将相邻的 I/O 请求做了合并而已。它也有很广泛的使用范围，比如有专门调度机制的存储设备，或是 SSD、NVRAM 这种快速存储，又或是 KVM 虚拟机镜像，这些都是使用 NOOP 算法的典型实例。

## 4. Anticipatory

3.X 版本以后的内核已经取消了 Anticipatory 算法。它本质上和 Deadline 很像，只不过多了一个 antic\_expire 的参数，表示在处理完最后一个请求后，调度器会再耽搁一会儿，说不定随后不久还会出现新的 I/O 请求。这种调度逻辑在连续读上的效果会比较好。

## 14.4.3 日志模式

ext4 是 CentOS 6 默认的文件系统，其默认挂载选项在绝大多数场景下可以达到最优的状态。从 ext3 开始就引入了日志的概念，日志模式总共有三种，分别是 journal、ordered 和 writeback。journal 模式会将全部数据都记入日志，落盘时必须先写日志，后写数据，确保了数据的高一致性。由于日志是有顺序的，写入速度较慢，所以它的写性能非常差。ordered 模式和 writeback 模式的日志都只记录 metadata，而且写数据时不用再等待日志的写入了，速度就会快很多。不同的是，前者先写数据，如果数据落盘但 metadata 日志没写上，那么这段数据就会被丢掉，也能保证数据的一致性。后者则不然，它把 metadata 日志

和数据彻底拆开了。例如，我用 vim 修改了一个文件，当 metadata 日志提交后，应用会认为文件已经更新了，但实际上数据可能还没有处理，如果此时系统发生 Crash，数据就不一致了。

默认的 ordered 和 writeback 的性能差异不大，没必要冒风险去修改日志模式。另外，我也不推荐将日志迁移到外部存储。这是一个很老的解决方案了，当时固态盘的售价比较昂贵，数据存储都是机械盘，所以才推荐使用 SSD 来存储日志部分。如果你的服务器全是 SSD 或者业务是以读取为主，则使用这个方案获得不了太多收益，而且单搞一块盘很浪费。如果使用机械盘且写入有压力，又不想增加太多成本，加一块 SSD 还是可取的。按照如下方法操作，可以完成日志存储的迁移。

以下是将 sdb1 上的日志迁移到 sdc1 上的过程，首先要确认原盘上的日志信息与块的大小，然后再根据相关信息初始化新盘，并完成日志的迁移工作。

```
# dumpe2fs /dev/sdb1 |egrep -i 'journal|size'
# tune2fs -O ^has_journal /dev/sdb1
# mke2fs -O journal_dev -b <Block-Size> /dev/sdc1
# tune2fs -j -J device=/dev/sdc1 /dev/sdb1
```

#### 14.4.4 其他因素

除了调度算法以外，queue 子目录下面还有很多其他可调的内核参数，它们都会影响队列深度的变化。

##### 1. 预读

在以顺序读为代表的应用场景中，预读技术会带来更好的用户体验。尤其是像音视频站点这种针对大文件的读取，因为你后面要读取的内容都是可预见的。这样，内核参数 read\_ahead\_kb 默认的 128KB 预设值就显得有些捉襟见肘了。适度提升 read\_ahead\_kb 可以有效提升顺序读取的性能。这里可以通过两种方式进行定义：设置 read\_ahead\_kb 可以预读多少 KB，或者使用 blockdev 命令定义允许系统预读多少个扇区（一个扇区是 512 字节）。设置其中任何一项，另外一项的数值都会跟随变化。

对于 Device Mapper 或者 RAID 之类的设备，通常应该增大该值，这是因为它们是由多块 PD 组成的，建议用默认值乘以 PD 的数量作为调整参数的起点为佳。

```
// 查看预读预设值
[root@station101 ~]# cat /sys/block/sda/queue/read_ahead_kb
128
[root@station101 ~]# blockdev --getra /dev/sda
256
// 下面这两个命令是等价的
# echo 256 > /sys/block/sda/queue/read_ahead_kb
# blockdev --setra 512 /dev/sda
```



## 2. Throughput、IOPS 和 Latency 之间的关系

Throughput、IOPS 和 Latency 是我们在 I/O 测试当中最为关注的三个重要指标。在第 5 章中我们已经讲述过 Throughput 和 IOPS 的概念了。Throughput 反映了磁盘每秒所能传输的数据量大小。它会受到 Block Size 以及读写模式的影响。Block Size 越大，Throughput 相应就会越高。而读、顺序、异步、有缓存相对要比写、随机、同步、无缓存要快得多，自然 Throughput 也就更高。

IOPS 指的是磁盘每秒进行了多少次 IO 操作。Block Size 是在文件系统初始化时定义的一个常量，而读写模式在应用场景的束缚下也是相对不变的。所以要想获得最大化的 Throughput，只有提升 IOPS 这一条路了。拿 Intel S3510 系列的固态硬盘举例来说，大部分 SSD（240GB 以上）随机写模式的 IOPS 都得上万，将近是机械盘的一百倍。这也就是为什么固态硬盘的性能会优于机械盘的原因了。

另外一个指标就是 Latency，这里所说的 Latency 特指 I/O 的 Latency，也就是一个 I/O 的响应时间，通常它的单位是毫秒或者微秒。Latency 包括了 I/O 处理和等待两部分，所以它的大小与 I/O Size 和队列深度都有关系。由于存储缓存的原因，系统可以合并多个 I/O，所以 I/O Size 的大小是不定的，但它依旧和 Block Size 成正比。另外一个因素就是当队列深度低于存储的处理能力时，通过增加队列深度可以改善 Latency。但如果调过了头，则会适得其反。

我们还是使用第 5 章中的那个例子来说明。假设饭馆大厨每小时能做 100 个菜，每个就餐客人平均会点 5 个菜，那么饭馆每小时的接待队列为 20 人时就是最佳状态。队列太短，大厨的工作量就不饱和。但如果队列太长，大厨忙不过来，肯定也不行。

内核参数 `nr_requests` 用来控制某块磁盘上允许分配的读写请求数量，适度提升该数值可能会改善磁盘 I/O 的吞吐能力，但这是以牺牲内存占用为代价的。该项操作对于小文件（Block Size  $\leq$  32KiB）且为 CFQ 算法的影响会是积极的。反之，对于 NOOP 和 Deadline 来讲，降低队列也许会更好一些。`nr_requests` 的默认值是 128。

```
[root@station101 ~]# cat /sys/block/sda/queue/nr_requests
128
```

## 3. Chunk Size 与 Stride

Chunk Size 是指 RAID 上一次写入的数据总量。可以借助 `iostat` 中的 `avgrq-sz` 来计算 Chunk Size，具体详情如公式 14-8 所示。`avgrq-sz` 指平均一次 I/O 操作所占的扇区数。通常 Chunk Size 的单位是 KiB/Disk。Data Disks 是指一个 I/O 操作产生时磁盘阵列中真正写入数据的盘数。例如，四块磁盘组成的 RAID 10 的 Data Disks 就应该等于 2。注意，Chunk Size 的值应为 2 的  $n$  次方。

$$\text{Chunk Size} = \text{avgrq-sz} \times 512 / 1024 / \text{Data Disks} \quad (14-8)$$

Stride 是指 Chunk Size 对应的 Block 数，而 Stride Width 则代表整个阵列上的 Block 数。Stride 和 Stride Width 的计算方式如公式 14-9 和公式 14-10 所示。

$$\text{Stride} = \text{Chunk Size} / \text{File System Block Size} \quad (14-9)$$

$$\text{Stride Width} = \text{Stride} \times \text{Data Disks} \quad (14-10)$$

取得 Stride 和 Stride Width 后，在初始化文件系统时可指定对应的参数。

```
# mke2fs -t <FS_TYPE> -b <FS_Block_Size> \
-E stride=<STRIDE>, stride-width=<STRIDE_WIDTH> <DEVICE>
```

#### 4. 其他参数

##### (1) max\_sectors\_kb

该参数用于控制发送到磁盘的最大 I/O 请求。它的最小值等于 Block Size，最大值不能超过 max\_hw\_sectors\_kb 的限制。如果发送请求的大小超过了磁盘内部定义的操作块的大小，会引发性能下降。此时需要降低 max\_hw\_sectors\_kb 的数值，在调整该数值之前，建议先完成针对不同 Block Size 的 I/O 测试工作。

```
[root@station101 ~]# cat /sys/block/sda/queue/max_sectors_kb
280
[root@station101 ~]# cat /sys/block/sda/queue/max_hw_sectors_kb
280
```

##### (2) rotational

rotational 只针对机械盘。如果你的存储是 SSD，在调整成 NOOP 算法后，应当将其设定为 0，防止调度器使用寻道逻辑。否则，在 SSD 设备中使用寻道操作会产生轻微的负效应。

```
[root@station101 ~]# cat /sys/block/sda/queue/rotational
1
```

##### (3) nomerges

这是一个调试用的选项，正常情况下它都应该处于禁用的状态。如果你要测试存储处理 IOPS 的能力，可以临时将它打开，并和禁用阵列卡的 Cache 一起使用。

```
[root@station101 Documentation]# cat /sys/block/sda/queue/nomerges
0
```

## 14.5 网络

我很难想象一台没有网络的服务器是什么样子的。想想也对，没有网络还能叫服务器吗？所以，网络性能同样是一个不可忽视的问题。然而，校准网络性能已经不再是一个纯系统层面的问题了。这需要 SE 具备一定的网络知识。

### 14.5.1 Jumbo Frames

IEEE 以太网帧的长度为 1518 字节，如果再加上 802.1q 的 TAG 长度是 1522 字节。事实上，结尾 4 字节长的 FCS 校验位所能校验的数据长度并不止这些。所以，Jumbo Frames 的概念就被提出了。它的 MTU 要比传统的 1500 字节更多，目前已知的最大值为 9000 字节。尽管 Jumbo Frames 并未被 IEEE 所认可，但其实很多厂商的设备都是支持的。

典型的 Jumbo Frames 应用场景为 NFS 和 iSCSI。过小的 MTU 使得系统必须提高发送频率才能满足应用需求。但是这样一来，数据包的封装、校检等任务量也大大增加，又导致了 TCP/IP 协议栈过度消耗 CPU 的资源。这时需要增加 MTU 的大小，MTU 既可以在 /sys 目录中可以临时调整，也可以在网卡配置文件中永久定义。

### 14.5.2 BDP

BDP 全称是 Bandwidth Delay Product，即带宽延迟乘积。顾名思义，它的数值就是用带宽和延迟的乘积得来的。它代表网络传输过程中留在队列中的数据大小。BDP 越大，说明网络延迟越高。延迟时间的测量非常简单，使用 ping 命令得到的 avg 就是平均的延迟时间。BDP 的计算公式如公式 14-11 所示。

$$\text{BDP} = \text{Bandwidth} \times \text{Delay} \quad (14-11)$$

应用程序是通过 Socket 进行网络请求和应答的，Socket Buffer 用来缓存接收过来的数据。这个 Buffer 的大小和我们前面提到的 BDP 有着很紧密的关联。rmem\_default 和 wmem\_default 这两个内核参数用来设定收发数据包时默认的 Buffer Size。rmem\_max 和 wmem\_max 对应的则是最大值。

TCP 协议可以利用滑动窗口自动协商调整。它有特定的内核参数可以设置。tcp\_rmem 和 tcp\_wmem，它们是由最小值、默认值和最大值三个数字组成的，而且最大值不能超过 rmem\_max 和 wmem\_max。默认情况下，rmem\_max 和 wmem\_max 的数值较小。对于一个延迟较高的网络而言，尤其是系统处于高并发的尖峰时段，有可能会不够用。而 tcp\_rmem 和 tcp\_wmem 的最大值是根据内存得来的，此时又被 rmem\_max 和 wmem\_max 所压制，性能就会受到很大的影响。因此，必须根据实际情况提升 rmem\_max 和 wmem\_max。这些参数的设定都取决于 BDP 的结果。

```
[root@station101 ~]# sysctl -a |grep -P "(_|\.) (r|w) mem"
net.core.wmem_max = 124928
net.core.rmem_max = 124928
net.core.wmem_default = 124928
net.core.rmem_default = 124928
net.ipv4.tcp_wmem = 4096      16384    4194304
net.ipv4.tcp_rmem = 4096      87380    4194304
net.ipv4.udp_rmem_min = 4096
```



```
net.ipv4.udp_wmem_min = 4096
```

### 14.5.3 qperf

第 5 章中我们介绍了网络测试工具 `iperf` 和 `netperf` 的使用，这里我再向大家介绍一款更加简单易用的工具——`qperf`。它可以很方便地测量出两个节点之间的网络带宽和延迟。这也是一个用来计算 BDP 的好方法。

假设，我们要测试 `station101` 和 `station103` 这两台服务器之间的链路带宽与延迟。首先在 `station103` 的后台运行 `qperf` 充当服务端，再在 `station101` 上以客户端模式对链路进行测试，最后可得到相应的结果。

```
// 服务端 station103 在后台运行 qperf
[root@station103 ~]# qperf &
[1] 39171
```

```
// 客户端 station101 测量 BDP
[root@station101 ~]# qperf station103.example.com tcp_bw tcp_lat conf
tcp_bw:
    bw = 118 MB/sec
tcp_lat:
    latency = 24.5 us
conf:
    loc_node = station101.example.com
    loc_cpu = 32 Cores: Intel Xeon E5-2650 v2 @ 2.60GHz
    loc_os = Linux 2.6.32-431.el6.x86_64
    loc_qperf = 0.4.9
    rem_node = station103.example.com
    rem_cpu = 32 Cores: Intel Xeon E5-2650 v2 @ 2.60GHz
    rem_os = Linux 2.6.32-431.el6.x86_64
    rem_qperf = 0.4.9
```

### 14.5.4 其他

除此之外，还有很多和网络性能相关的技术点。比如在第 4 章中提到的网卡 Bonding 模式，我们建议大家优先使用 802.3ad 或 ARP 协商这两种模式。还有第 5 章中的零拷贝技术，它可以有效地减少系统开销，从而提升传输效率。如果业务增长速度很快，建议优先使用万兆网络，以免在后期出现瓶颈。针对单一网卡队列或虚拟网卡，启用 RPS 和 RFS 可将数据流分布在指定的 CPU 核心上，防止因软中断集中落在某个繁忙的核心上而导致的网络性能下降。

网络栈在很大程度上是自我优化的。另外，有效校准网络性能要求我们对网络栈要有深入的理解。盲目、单调地调整不但达不到目的，反而会适得其反。鉴于笔者水平有限，广大读者如有兴趣可自行查阅相关资料。

## 14.6 本章小结

本章我们讨论了和系统性能相关的一些知识。和其他章节不同的是，我们一直无法就性能调优这个主题给出一个具体的实施方案。具体原因我们在开篇中已经提到了。这里还要特别说明的一点是，本章重在理论讲解，请大家在实际应用当中一定要用充分的测试数据来说话。每次测试只确定一个目标，只调整一个参数，结合队列理论并使用正确的测试方法，这样得到的结论才是可信的。下一章将是本书技术环节的最后一章。届时，我将和大家一同探讨 Shell 编程的话题。

## 第 15 章

# Shell 编程

严格地说，Shell 算不上是一种语言，但它却是所有语言里最适合做系统管理的。关于 Shell 的缺点，我完全能列举出一大堆例子来。比方说不支持面向对象、弱类型、执行效率低、无法跨平台，等等。这些都不重要，本来 Shell 也不是用来编写大型程序的，更没打算和谁去比执行效率，但它有一个其他语言都无可替代的优势——直观。系统管理依赖于命令，而 Shell 在编写代码时基本上保留了 SE 所有的原始操作习惯。只加入了少量流程控制，就能很好地将原来孤立的命令连接在一起。

我曾经学习并使用过很多种开发语言。不过就脚本语言来讲，我还是最喜欢 Shell。这也许与我本身是一名 SE 有些关系吧。关于开发语言的选择，可能每个人都有着不同的见解。我比较偏重于语法、函数库和效率这三个方面。

语法这东西就跟颜值一样，得看眼缘儿。可能有人就看不惯用空格标记代码层次的做法，也有人不喜欢语句后面啰唆的分号，就连函数体的花括号应该放在哪里都有很多的争议。由于语法上的好恶，你一看它就别扭，肯定就不愿意去学习它了。

对于一个 SE 来说，Shell 比任何一门语言都有亲和力。90% 以上都是 SE 熟悉的系统命令，几乎没有什么学习成本，上手很容易。编写代码时甚至都不必过多地思考，心里怎么想，手上就不自觉地敲出来了。从语法的角度上看，Shell 跟 SE 在一块儿特别有“夫妻相”。

光注重外表还不够，我们也得考察一下内在美。函数库就是这样一个有内涵的东西，它体现了语言的原生能力。学习一门开发语言，语法的占比很小，大部分精力还是要花在对函数库的研究上。如果掌握不好函数库的使用，光懂语法是不行的。同样一件事，别人两下就干好了，而你了解函数库，就需要不断地摸索和尝试。从了解到熟练掌握，这需要一个非常漫长的过程。

而 Shell 是高度封装过的，就跟乐高积木一样。你不用着学习什么函数库，每一条指令都是一个实例化的函数，方法是选项，属性是参数，只需要你将它们组装起来就是了。而且指令要比函数库丰富得多，光 yum 源中提供的软件包少说也得有几千种。更妙的是，指令的学习不像函数库，非要放到代码中去执行。如果这样你还要抱怨 Shell 的功能弱，那只





能说你的命令行功底还不过关。

最后一个因素就是效率。这里说的效率和执行效率不同。有些人就特别喜欢拿执行效率来说事儿，说 Shell 执行速度慢。要讲快，汇编最快了，那我们为什么还要有这么多种不同的开发语言呢？

谈执行效率要看怎么讲。首先是规模。如果你的代码就几千行，你能比我快多少？语言的性能优势又能体现在哪里？其次是业务的容忍度。比方说，同样一个活儿，可以用 A、B 两种语言，执行时间上 A 比 B 快半分钟，但 B 比 A 更容易维护。你说我们选谁？我说要看总体效率，要分清主次。如果业务不接受延迟就让步给 A，这时候延迟就是总体效率。但如果延迟在可控的范围内，业务能够接受就选 B。因为你要考虑 A 的维护成本啊。改一个代码，A 要三天，而且只有两个人会改，B 要两天，有六七个人都能胜任。这时的总体效率就变成维护成本了，维护周期越短效率就越高。

没有必要再去争论什么语言的优劣，一门语言只要能存活下来，就一定有它的优势所在。世界上只有烂代码，没有烂语言。还是那句话，语言没有三六九等之分，只有合适与否之别。什么叫合适？合适就是性价比最好、综合得分最高的那个。

如果你想要完成一项系统管理的工作，没有比 Shell 更适合、更便捷的语言了。只要是指令能完成的任务，你又何必舍近求远去用其他语言来实现呢？在这一章中，我们将围绕着 Shell 中的核心部分，对其中难以理解的地方进行深入剖析，并通过完整的示例代码讲解它们的实现方法。

## 15.1 参数传递

一个程序从运行到消亡，它这一生总共就只干了三件事情——数据接收、数据处理和数据输出。所谓的数据接收就是指参数传递了。大部分 Linux 指令都是由命令、选项和参数所组成的。参数传递是与用户交互的入口，也是保证程序能够正常运行的关键一步。本节将为大家介绍如何解决参数传递中的问题。

### 15.1.1 shift

shift 的作用是参数位移。众所周知，Shell 使用的是位置参数。位置参数采用 `${NUMBER}` 的形式来表达。例如，`$1` 就代表第一个参数。位置参数可以传递多个，由于位置是固定的，如果我们需要分批处理，没有 shift 的帮助，实现起来会非常困难。

例如，程序要传递  $N$  个参数，每次只处理前两个。这个需求如果我们要自己来构造，该如何实现呢？我想应当是这样一个过程：首先计算  $N/2$ ，然后循环  $N/2$  次，每次循环位置参数 +2。假如用户提交了 6 个参数，总共需要循环 3 次，按照 `$1 $2`、`$3 $4`、`$5 $6` 分成三组来处理。

但你要表达这三组位置参数时，会有点儿困难。因为位置参数是变化的，在循环体中，我们分别要用 \$x 和 \$y 来表示前两个位置参数。在三次循环中，x 和 y 的取值分别是这样的。

```
当 loop=1 时, x=1; y=2;
当 loop=2 时, x=1+2=3; y=2+2=4;
当 loop=3 时, x=3+2=5; y=4+2=6;
```

我们看，你需要通过运算来改变 x 和 y 的值，必须增加两个运算表达式。另外，在第一次循环时，x 和 y 是直接赋值的。这说明表达式并不唯一，到底是赋值还是计算，还得增加一层判断才行。也就是说，在循环体里面还要有一个分支结构，这太复杂了。

看到这里，有朋友可能会说：既然如此，那我改进一下，给 x 和 y 初始化赋值不就行了么？但是你有没有考虑到，当我的需求变更成每次处理三个参数时，你还要重新计算 x 和 y 的值，并修改表达式，这就显得有些麻烦了。

下面就是改进后的方案，但当需求变更后，修改成本依旧很高。

```
初始化, x=-1; y=0;
当 loop=1 时, x=-1+2=1; y=0+2=2;
当 loop=2 时, x=1+2=3; y=2+2=4;
当 loop=3 时, x=3+2=5; y=4+2=6;
```

假如我们直接用 shift，那就省心多了。它实现了位置参数的左移操作。例如，shift 2 代表将所有位置参数向左移动两位。第一次循环开始时，原先的 \$1~\$2 因为左移的缘故被删掉了，而 \$3~\$6 变成了新的 \$1~\$4，后面以此类推。此时前两个位置参数可以直接写成 \$1 和 \$2，因为它们的值会随着左移而不断地变化。

如果说参数是包裹，循环体内的取值操作是机械臂，那么 shift 就像是一个向左移动的传送带。我们以前取值，包裹是固定的，只能依靠移动机械臂来抓取。但自己构造机械臂的移动逻辑太吃力了。而当我们把包裹放到传送带上，机械臂无须移动，直接抓取就可以了。这样是不是就简单多了？

使用 shift 实现参数位移的示例代码如下。

```
[root@station103 ~]# cat -n shift.sh
1  #!/bin/bash
2
3  loop=`expr $(echo $#) / 2`
4
5  for i in $(seq 1 $loop)
6  do
7      echo $1 $2
8      shift 2
9  done
10
```

// 执行结果如下，由于传递参数不是偶数，所以最后一个参数 7 在做除法时被丢弃了

```
[root@station103 ~]# sh shift.sh 1 2 3 4 5 6 7
1 2
3 4
5 6
```

### 15.1.2 eval

好，获取位置参数的问题解决了，接下来我们再聊聊获取变量值的问题。

获取变量的值需要使用 `echo` 命令。但如果变量的值不是数字或字符串，而是一条命令，并且我们还要去执行它，以实现类似 Ansible 的 Ad-Hoc 或者 SaltStack 的 `cmd.run`，就需要派遣 `eval` 出场了。

`eval` 的原理就是先读取变量的值，然后再执行，类似于下面这样的效果。

```
echo $VARIABLE |bash
```

下面这段示例代码很好地说明了 `eval` 的作用。它是不是很像一个简化版的 SaltStack？它将第一个位置参数中的内容当作命令去执行，并返回该命令以及输出结果。

```
[root@station103 ~]# cat -n eval.sh
1  #!/bin/bash
2
3  cmd=$1
4  echo command: $cmd
5  echo results:
6  eval $cmd
```

// 执行结果如下

```
[root@station103 ~]# sh eval.sh pwd
command: pwd
results:
/root
```

// 等同于类似的效果

```
[root@station103 ~]# echo pwd |bash
/root
```

细心的读者朋友可能会发现，其实去掉 `eval` 后，结果也是一样的。既然如此，这个 `eval` 有什么特别之处呢？看完下面这两个例子后，读者朋友们就能明白了。这次我们换一条命令，并使用选项 `-x` 将执行过程打印出来。

// 这是使用 `eval $cmd` 的执行结果

```
[root@station103 shell]# sh -x eval.sh "grep ^root /etc/shadow |cut -d: -f1"
+ cmd='grep ^root /etc/shadow |cut -d: -f1'
+ echo command: grep '^root' /etc/shadow '|cut' -d: -f1
command: grep ^root /etc/shadow |cut -d: -f1
+ echo results:
results:
+ eval grep '^root' /etc/shadow '|cut' -d: -f1
```



```

++ grep '^root' /etc/shadow
++ cut -d: -f1
root
// 这是直接使用 $cmd 的执行结果
[root@station103 shell]# sh -x eval.sh "grep ^root /etc/shadow |cut -d: -f1"
+ cmd='grep ^root /etc/shadow |cut -d: -f1'
+ echo command: grep ^root /etc/shadow |cut -d: -f1
command: grep ^root /etc/shadow |cut -d: -f1
+ echo results:
results:
+ grep ^root /etc/shadow |cut -d: -f1
grep: unknown directories method

```

通过比较二者 `results` 后面的结果，我们看出了它们之间的区别。`eval` 能够识别管道符，并将命令拆成两部分执行，而 `$cmd` 却把管道符给转义了，导致后面的 `-d:` 也变成了 `grep` 的参数，但 `grep` 无法识别它，所以返回了错误。

```

// 模拟等价的错误用法
[root@station103 ~]# grep -d: /etc/shadow
grep: unknown directories method

```

接下来，我们再来看一个 `awk` 的例子。

```

// 这是使用 eval $cmd 的执行结果
[root@station103 shell]# sh -x eval.sh "awk -F: '/^root/ {print \$1}' /etc/shadow"
+ cmd='awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow'
+ echo command: awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow
command: awk -F: '/^root/ {print \$1}' /etc/shadow
+ echo results:
results:
+ eval awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow
++ awk -F: '/^root/ {print \$1}' /etc/shadow
root

// 这是直接使用 $cmd 的执行结果
[root@station103 shell]# sh -x eval.sh "awk -F: '/^root/ {print \$1}' /etc/shadow"
+ cmd='awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow'
+ echo command: awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow
command: awk -F: '/^root/ {print \$1}' /etc/shadow
+ echo results:
results:
+ awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow
awk: '/^root/
awk: ^ invalid char '"' in expression

```

显然，`$cmd` 又做了多余的事情，这次它转义的是单引号，其效果相当于下面这种错误的用法。由于语法错误致使 `awk` 无法执行。

```

// 模拟等价的错误用法
[root@station103 ~]# awk -F: '\ '/^root/ {print \$1}' '\ ' /etc/shadow

```



```
awk: '/^root/  
awk: ^ invalid char ''' in expression
```

### 15.1.3 getopt

前面我们已经指出，系统指令通常是由命令、选项和参数三部分组成的。命令是执行的动作，选项用来描述命令的执行方式，而参数则代表被操作的对象。我们举个例子来说明——老师严厉地批评了小明。在这里，“批评”是命令，“严厉地”是选项，“小明”是参数。

尽管选项和参数并不是必需的，但大多系统指令都离不开它们。用户需要根据当前情况，去做出不同的选择。然而，用户一旦参与输入，就会带来无尽的麻烦。比方说：

- ❑ 如果用户把很多个选项写在一起了怎么办？
- ❑ 如果用户漏掉了必须输入的部分怎么办？
- ❑ 如果用户输入的内容是错误的怎么办？
- ❑ 如果用户把参数顺序搞乱了又该怎么办？

上述这些问题我们都可以统称为“语法错误”。语法检查是必需的，但如果要我们自己去构造这些逻辑是非常困难的。还好我们有现成的 `getopt` 模板可以直接套用。它只要我们把自定义参数填进去就可以了，剩下的事情都交给 `getopt` 去处理。

`getopt` 是被用来解析选项语法的，再配合上 `set` 命令，我们自己的 Shell 程序也可以和系统指令一样——实现诸如语法分析检查、允许短选项合并、无视参数顺序等强大的功能。

选项分为短选项和长选项两种。短选项使用一个中横线开头，后面紧跟单个字符。长选项则使用两个中横线开头，后面紧跟一个词。短选项可以有条件的合并，而长选项却不可以。接下来，我将借助两段不同的代码，分别为大家介绍它们的实现方式。

#### 1. 短选项

下面这段代码是一个短选项的 `getopt` 实例。

```
[root@station103 ~]# cat -n getopt1.sh  
1  #!/bin/bash  
2  
3  OPT=`getopt -o ab:c:: -- "$@"`  
4  
5  eval set -- "$OPT"  
6  while true ; do  
7      case "$1" in  
8          -a) echo "Option a" ; shift ;;  
9          -b) echo "Option b, parameter \"$2\"\" ; shift 2 ;;  
10         -c)  
11             case "$2" in  
12                 "") echo "The parameter of Option c is missing.\" ; shift 2 ;;  
13                 *) echo "Option c, parameter \"$2\"\" ; shift 2 ;;
```

```
14         esac ;;
15     --) shift ; break ;;
16     *) echo "getops error!" ;;
17     esac
18 done
```

第 3 行是整段代码当中最核心的部分。getopt 由 optstring 和 parameters 组成，也就是我们所说的选项和参数。请注意，尽管 -o 不是必需的，但如果执行命令时找不到它，getopt 会将第一个参数当作短选项使用，这可能有违我们的初衷。-- 在 bash 中代表一个选项的结束，位于它后面的所有内容，都将被当作参数部分。它是用来“显式分隔”optstring 和 parameters 的。-o 与 -- 之间的内容就是自定义选项。这里我们自定义了 a、b、c 三个选项。字母后面没有冒号代表它是一个独立的选项，不需要参数跟随。后面有一个冒号代表选项需要一个参数，参数和选项之间可以有一个空格，也可以连接在一起。后面有两个冒号代表选项需要一个参数，且参数必须紧跟在选项后面。

关于第 5 行中 eval 的作用，想必大家应该已经很清楚了。它将把 getopt 的整个内容代入 set 命令中去执行，最后由 set 完成选项和参数的传递工作。

第 6~18 行是一个无条件的循环体。因为我们可能需要接收多个参数，所以该循环不会自己结束。getopt 要配合 case 语句，根据不同的参数来执行不同的动作。为了演示方便，我在这里执行的是 echo 命令。实际应用中，应当使用函数调用替换掉这里的 echo 语句。

由于是无条件循环，所以我们需要设置结束条件。第 15 行表示，当所有的选项和参数都用光后，while 循环将会结束。而 shift 就是用来移除那些使用过的选项和参数的。

理解了上述这些内容，我们来看一下执行结果。

```
[root@station103 ~]# sh getopt1.sh -a
Option a
[root@station103 ~]# sh getopt1.sh -b1
Option b, parameter '1'
[root@station103 ~]# sh getopt1.sh -b 1
Option b, parameter '1'
[root@station103 ~]# sh getopt1.sh -c1
Option c, parameter '1'
[root@station103 ~]# sh getopt1.sh -c 1
The parameter of Option c is missing.
[root@station103 ~]# sh getopt1.sh -abc -c1
Option a
Option b, parameter 'c'
Option c, parameter '1'
```

最后一条指令中，选项 abc 合并后，合并进去的 c 实际上被当成选项 b 的参数来使用了。所以带有参数的短选项是不能合并的，否则会产生语法歧义。日常执行命令时，也应当注意这个问题。



最后一条指令如果在执行时加上 `-x`，我们可以从扩展信息中看到如下内容。

```
[root@station103 ~]# sh -x getopt1.sh -abc -cc|head -4
++ getopt -o ab:c:: -- -abc -cc
+ OPT=' -a -b '\''c'\'' -c '\''c'\'' --'
+ eval set -- ' -a -b '\''c'\'' -c '\''c'\'' --'
++ set -- -a -b c -c c --
...
```

原来 `getopt` 的整个执行过程就是——解析我们定义好的选项和参数，并以空格形式拆分，最后提交给 `set` 命令完成传递工作。

## 2. 长选项

当我们理解了短选项的代码示例后，再去实现长选项就容易多了。除了使用 `--long` 来标识一个长选项，且选项之间要用逗号分隔以外，其他部分全部照搬即可。使用长选项需要注意折行问题，两个长选项之间是不能有空格的。程序员为了代码的可读性，大多会采用“\”来编辑多行内容。如果长选项的部分需要折行，请一定要确保新行是顶头书写的。

下面这段代码就是一个长选项的 `getopt` 实例。

```
[root@station103 ~]# cat -n getopt2.sh
1  #!/bin/bash
2  TEMP=`getopt -o h\
3      --long name:,image:,cpu:,memory: \
4      -- "$@"`
5
6  eval set -- "$TEMP"
7  while true ; do
8      case "$1" in
9          -h) break ;;
10         --name) NAME=$2; shift 2;;
11         --image) IMAGE=$2; shift 2;;
12         --cpu) CPU=$2; shift 2;;
13         --memory) MEMORY=$2; shift 2;;
14         --) shift ; break ;;
15         *) echo "getops error.";;
16     esac
17 done
18
19 echo NAME: $NAME
20 echo IMAGE: $IMAGE
21 echo CPU: $CPU
22 echo MEMORY: $MEMORY
```

通过执行代码，我们看到了两个有趣的现象。

首先，我们不必关心选项的顺序。很多初学者在编写 Shell 时，都习惯使用位置参数。所以，他们的程序通常只能写给自己用，却很难放心地交给别人去执行。位置参数在执行时，顺序一旦颠倒，执行结果将难以预料。而 `getopt` 却能处理得很好。长选项表达清晰，

起到了提示用户的作用，执行人很清楚选项后面要输入的内容是怎样的。另外，由于选项和参数是一一对应的，输入顺序也就无关紧要了。

其次，长选项不一定非要写完整。只要能够唯一标识，不写全是也可以的。如果标识不唯一，getopt 会选择源码中第一个匹配的长选项。

```
[root@station103 ~]# sh getopt2.sh --cpu=4 --mem=8192 --na=mydocker --im=centos6
NAME: mydocker
IMAGE: centos6
CPU: 4
MEMORY: 8192
```

### 15.1.4 函数传参

尽管不支持面向对象，可我们还是可以使用函数来书写 Shell 程序。函数传参也是用位置参数来实现的。上一节中，我们用 getopt 实现了指令语法的规范化，不过这里还有一点缺陷——我们没有对“空对象”做检查。

某些指令必须要指定操作对象，例如 cp 命令，关键的参数是不能缺失的。另外在不同情况下，关键参数亦会随之变化。拿 Docker 来说，创建容器时至少要提供一个镜像文件，而启动容器时则必须指定容器的名称或 ID。getopt 能够完成选项的语法检查，但如果用户压根儿就没有使用某一个选项，使得某个关键参数缺失，则将导致命令执行失败。

下面这段示例代码综合应用了之前的几个例子，用来检查对象是否为空。

```
[root@station103 ~]# cat -n null.sh
1  #!/bin/bash
2  TEMP=`getopt -o h\
3      --long name:,image:,cpu:,memory: \
4      -- "$@"`
5
6  eval set -- "$TEMP"
7  while true ; do
8      case "$1" in
9          -h) break ;;
10         --name) NAME=$2; shift 2;;
11         --image) IMAGE=$2; shift 2;;
12         --cpu) CPU=$2; shift 2;;
13         --memory) MEMORY=$2; shift 2;;
14         --) shift ; break ;;
15         *) echo "getops error.";;
16     esac
17 done
18
19 func_nullCheck()
20 {
21     loop=`expr $(echo $#) / 2`
22     for i in $(seq 1 $loop)
23     do
```

```
24     if [ -z "$2" ]; then
25         echo $1 is null.
26         case "$1" in
27             name) return 20;;
28             image) return 21;;
29             cpu) return 22;;
30             memory) return 23;;
31             *) return 29;;
32         esac
33     else
34         shift 2
35     fi
36 done
37 }
38
39 func_nullCheck name "$NAME" image "$IMAGE" cpu "$CPU" memory "$MEMORY"
```

前半部分依旧是一个 `getopt` 的例子，后半部分利用 `shift` 完成了对第二个位置参数是否为空的检查。第 39 行定义了该条指令所需要的关键参数有哪些。如果它们没有被找到，`func_nullCheck()` 会提示用户对象缺失。利用长选项可以将选项和参数绑定在一起，不必担心用户在输入时会丢失关键信息。

```
// 执行结果如下，由于 func_nullCheck() 检测到选项 --cpu 为空，于是返回错误。
[root@station103 ~]# sh null.sh --na=mydocker --im=centos6 --mem=8192
cpu is null.
[root@station103 ~]# echo $?
22
```

### 15.1.5 返回值

如果一个程序没有返回值是令人崩溃的。终端用户也许并不会特别在意，但它对于程序的逻辑判断却非常关键。

`return` 和 `exit` 都能够给予一个返回值，但它们之间是有区别的。`return` 用于函数内部，它在执行后会退出当前函数，并将返回值和控制权交给调用函数的对象。`exit` 在执行后会结束整个程序，并将返回值和控制权交还给操作系统。

使用 `echo $?` 可以获取最近一次的返回值，取值范围在 0~255 之间。如果返回值在设定时大于 255，获取返回值时，会求取该数值除以 256 的余数。返回值的设定是有规范的。0 代表成功，126 表示文件不可执行、127 说明命令尚未找到，128 及以上意味着系统产生了一个信号。在自定义错误的返回值时，应当使用 1~125 这个区间。即便如此，自定义也有一些约定俗成的规矩。比如，1 表示执行失败或语法错误，2 表示对象无法找到，等等。我们在写程序的时候，应当尽可能地按照规范去自定义返回值。

以下是我模拟的一些返回值的实例，仅供读者参考。

```
[root@station103 ~]# pwd
```



```
/root
[root@station103 ~]# echo $?
0

[root@station103 ~]# ./install.log
-bash: ./install.log: Permission denied
[root@station103 ~]# echo $?
126

[root@station103 ~]# lll
-bash: lll: command not found
[root@station103 ~]# echo $?
127

[root@station103 ~]# pwd^C
[root@station103 ~]# echo $?
130

[root@station103 ~]# ls sfd sdf
ls: cannot access sfd sdf: No such file or directory
[root@station103 ~]# echo $?
2

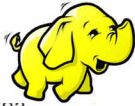
[root@station103 ~]# su - se
[se@station103 ~]$ cat /etc/shadow
cat: /etc/shadow: Permission denied
[se@station103 ~]$ echo $?
1
```

下面这段代码是一个返回值的实例，我们可以通过它来了解 return 和 exit 的用法。

```
[root@station103 ~]# cat -n return.sh
1  #!/bin/bash
2
3  func_return()
4  {
5      if [ "$1" == "$2" ]; then
6          return 0
7      else
8          return 1
9      fi
10 }
11 if [ -z "$1" ] || [ -z "$2" ]; then
12     echo 'parameter missing.' && exit 2
13 fi
14 func_return $1 $2
15 echo $?
```

// 执行结果如下

```
[root@station103 ~]# sh return.sh 1 1 ; echo $?
0
```



```
0
[root@station103 ~]# sh return.sh 1 2 ; echo $?
1
0
[root@station103 ~]# sh return.sh 1 ; echo $?
parameter missing.
2
```

由于系统返回值的取值范围较小，可能有时候会不够用。这需要额外构造一个外部函数，让它来承担向应用提供返回值的工作。进而产生了如何将自定义返回值交付给外部函数的问题。

例如，我们要创建一个 Docker 容器。根据业务需求，创建过程中必须提供 CPU、内存、镜像、容器名称等各种参数。为了防止在创建过程中缺失关键的参数，我们定义了一个参数缺失的错误集合 2XXXX。

为了使用这个自定义返回值，我们单独构造了一个外部函数 `send_status()`。它的语法格式如下，变量 `VALUE` 的取值范围支持 0~99 999。

```
send_status "RETURN_CODE:VALUE"
```

向 `send_status()` 发送返回值的内部函数如下所示。

```
func_return()
{
    if [ $? -eq 0 ]; then
        send_status "RETURN_CODE:0"
    else
        send_status "RETURN_CODE:$1"
        return 1
    fi
}
```

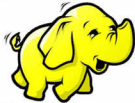
上一小节中，我们已经介绍了如何检查对象是否为空。我们可以根据 `func_nullCheck()` 中的情况来决定 `func_return()` 发送给 `send_status()` 的返回值。

实现起来很简单，找到 `func_nullCheck()` 中的第 27~31 行，将下面这行替换掉 `return` 语句即可。注意：2XXXX 要根据实际情况定义。例如，20001: CPU 参数缺失，20002: 内存参数缺失，以此类推。

```
test 1 -ne 1 && func_return "2XXXX"
```

## 15.2 文本处理三剑客

说完了数据接收，我们再来谈谈数据处理。除了 I/O 读写以外，我想这是程序中最耗资源的部分了。而数据处理又可以分为文本和计算两大部分。本节我们就先聊聊文本处理的话题。



## 15.2.1 grep

grep (Globally search a Regular Expression and Print) 是一个强大的文本搜索工具，可以根据指定模式来匹配并输出符合要求的字符串。

### (1) 统计匹配行数

这个功能很有用，比如它可以获取当前可用的处理器数量，我们就没必要再使用 wc 来计算了。但要注意，这里计数是针对一个文件对象而言的，如果文件对象有多个，每个对象都会单独统计。

```
[root@station103 ~]# grep -c processor /proc/cpuinfo
32
[root@station103 ~]# grep -c ^root /etc/passwd*
/etc/passwd:1
/etc/passwd-:1
```

### (2) 忽略大小写

Linux 系统对大小写是敏感的，如果不是特别确定，最好能够忽略大小写，以防疏漏。

```
[root@station103 ~]# grep -i ^port /etc/ssh/sshd_config
Port 22
```

### (3) 行号

如果我们需要在特定行附近做修改，首先要取得行号才行。

```
[root@station103 ~]# grep -n ^Port /etc/ssh/sshd_config
13:Port 22
```

### (4) 搜索周边

如果要搜索字符串附近的内容，可以使用如下几种方式。

❑ A{NUM}, A 代表 After, 打印匹配的行及其下面 X 行。

❑ B{NUM}, B 代表 Before, 打印匹配的行及其上面 X 行。

❑ C{NUM}, 打印匹配的行及其上面 X 行和下面 X 行。

比如，我要搜索以 Port 开头的字符串附近的内容，可以执行如下命令。

```
[root@station103 ~]# grep -C1 ^Port /etc/ssh/sshd_config
```

```
Port 22
```

```
#AddressFamily any
```

```
[root@station103 ~]# grep -B1 ^Port /etc/ssh/sshd_config
```

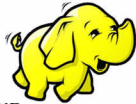
```
Port 22
```

```
[root@station103 ~]# grep -A1 ^Port /etc/ssh/sshd_config
```

```
Port 22
```

```
#AddressFamily any
```





## 1. 静默模式

有时我们只想知道指定的字符串是否有匹配，但并不关心具体的输出。选项 `-q` 表示进入静默模式，它不会产生输出，通过返回值表示匹配结果。返回值为 0 说明有匹配对象，否则就为 1。静默模式在代码中常会用到，它可以避免屏幕上的无效输出。

```
[root@station103 ~]# lsmod |grep kvm
kvm_intel          54285  18
kvm                333172  1 kvm_intel
[root@station103 ~]# lsmod |grep kvm -q
[root@station103 ~]# echo $?
0
```

## 2. 匹配的文件对象

如果想要了解哪些文件含有匹配的字符串，应当使用选项 `-l` 来实现。

```
[root@station103 ~]# grep -l ^root /etc/*
/etc/group
/etc/group-
/etc/gshadow
/etc/gshadow-
/etc/passwd
/etc/passwd-
/etc/services
/etc/shadow
/etc/shadow-
/etc/sudoers
```

## 3. 递归目录

如果要在某个目录下搜寻连同子目录的所有对象，需要使用递归选项 `-r`，就像下面这样。

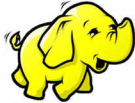
```
[root@station103 ~]# grep -r station103 /etc
/etc/sudo-ldap.conf:uri ldap://station103.example.com
/etc/sysconfig/network-scripts/ifcfg-em3:DHCP_HOSTNAME="station103.example.com"
/etc/sysconfig/network-scripts/ifcfg-em4:DHCP_HOSTNAME="station103.example.com"
/etc/sysconfig/network:HOSTNAME=station103.example.com
```

## 4. 包含与排除

如果目录下的文件对象太多，可以使用限定条件来约束，以削减无关内容。

- ☐ `--include=GLOB;`
- ☐ `--exclude=GLOB;`
- ☐ `--exclude-from=FILE;`
- ☐ `--exclude-dir=DIR.`

举个例子，下面这条命令用于找出 `/etc/` 目录下开头含有 `root` 的文件，但不包括以 “-” 号结尾的。



```
[root@station103 ~]# grep -l ^root --exclude=*- /etc/*
/etc/group
/etc/gshadow
/etc/passwd
/etc/services
/etc/shadow
/etc/sudoers
```

## 5. 高亮标记

如果返回结果爆屏，我们称之为信息泛洪。比如像我这种上了年纪、眼神儿不济的人来说，使用 `--color` 做高亮输出是必不可少的。匹配文本用红色标记，而匹配文件名将以紫色显示。

## 15.2.2 sed

`sed` 全称是 Stream Editor，正如其名，它是按照行的方式来处理文本的，主要用来实现内容的修改。`sed` 可以实现批量文本的替换、插入、修改和删除等操作。

### 1. 替换

为了便于演示说明，我们首先创建了一个示例文本，内容如下所示。这是一个单词本，里面有大量的目标词汇 `bird`。我们的目的是将 `bird` 替换成 `cat`。

```
[root@station103 ~]# cat -n text
1 bird work good seek
2 mark bird bird rest
3 push deep bird bush
4 desk jeep need year
5 beat meet seat bird
```

为了便于演示，我们不对示例文件做真正的修改。如果你确定要修改文件内容，请使用选项 `-i`。不添加 `-i` 的话，`sed` 只会将修改结果 `dump` 到屏幕上。

一旦文件内容被修改，将无法恢复，所以最好使用如下命令进行备份操作。

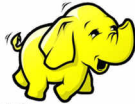
```
// 在修改 INPUT-FILE 之前，会生成一个名为 INPUT-FILE.bak 的备份文件
# sed -i.bak EDIT-COMMAND INPUT-FILE
```

文本替换的语法格式如下所示。

```
[Address]s/Pattern/Replacement/FLAGS
```

Address 可以是数字或字符，可以是某一行，也可以是一个范围，为空则代表整个文档。如果 Flags 是数字，表示替换第 X 个匹配对象，为空的话就是 1。如果 Flags 是 g，则会替换所有匹配的对象。

```
# sed "3s/bird/cat/2" text // 将第 3 行中的第二个 bird 替换成 cat
# sed "1,3s/bird/cat/g" text // 将前 3 行中的 bird 替换成 cat
# sed "/work/s/bird/cat/" text // 将含有 work 一行的第一个 bird 替换成 cat
```



```
# sed "/work/,/push/s/bird/cat/g" text // 找到含有 work 和 push 行, 将这些行之间的 bird  
替换成 cat
```

## 2. 插入、修改和删除

除了替换, sed 还可以实现插入、修改和删除的功能。

```
# sed "3i\cat" text // 在第 3 行上方插入 cat, cat 将作为新的第 3 行  
# sed "3a\cat" text // 在第 3 行下方插入 cat, cat 将作为新的第 4 行  
# sed "3c\cat" text // 将第 3 行替换成 cat  
# sed "3d" text // 删除第 3 行  
# sed "3r /root/text2" text // 在第 3 行下方将另外一个文件 text2 的内容追加进去
```

从上面的示例中我们看到, 内容插入只能从新的一行开始。如果要在行首行尾的插入操作, 还是要用到替换功能。替换位置中的 & 表示 Pattern 匹配到的内容。由于 .\* 表示所有, 即一整行, 所以我们在 & 前后写入的内容就等于插入到了行首或行尾。

```
# sed "3s/.*/cat &/" text // 在第 3 行行首插入单词 cat  
# sed "3s/.*/& cat/" text // 在第 3 行行尾插入单词 cat
```

### 15.2.3 awk

不可否认, awk 是 Linux 平台功能最强大的数据处理引擎。它提供了模式匹配、数学运算、流程控制、内置变量和函数等诸多强大的功能。它不仅仅是一个文本处理工具, 同时还是一门程序设计语言。由于它过于强大和复杂, 坊间流传过这样一种说法: 一个 Shell 程序员对于 awk 的理解深度, 甚至可以决定其代码水平的高低。

尽管 awk 的语法十分复杂, 但它的本质始终还是模式 + 行为。一条完整的 awk 语句包含了 BEGIN、END 和中间三部分。BEGIN 部分用于变量的初始化与赋值, END 部分多用于输出结果。这两部分并不是必需的, 只有当出现较为复杂的处理逻辑时, 我们才会使用到它们。

#### 1. 统计数量

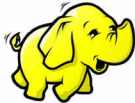
awk 可以使用自增运算来构造计数功能。和 grep 不同, awk 的统计功能是开放的, 它可以实现针对各种操作的计数。下面这个示例展示的是统计匹配关键字 git 的行数。

```
[root@station103 ~]# awk /git/ /etc/shadow  
gitlab-www!!!:17379:!!!!:  
git!!!:17379:!!!!:  
gitlab-redis!!!:17379:!!!!:  
gitlab-psql!!!:17379:!!!!:  
gitlab-prometheus!!!:17379:!!!!:  
[root@station103 ~]# awk '/git/{n++} END {print n}' /etc/shadow  
5
```

#### 2. 获取某一行

NR 是 awk 的内置变量, 代表行号, 如果要提取特定行, 可以用它配合条件语句 if 来





实现。

```
[root@station103 ~]# ps -C qemu-kvm
  PID TTY          TIME CMD
 13588 ?            00:03:28 qemu-kvm
 13638 ?            00:05:19 qemu-kvm
 13678 ?            00:05:12 qemu-kvm
 13727 ?            00:05:26 qemu-kvm
 29357 ?            3-23:02:45 qemu-kvm
// 使用 NR 变量提取上述输出的第 6 行
[root@station103 ~]# ps -C qemu-kvm |awk '{if(NR==6) print $0}'
29357 ?            3-23:02:45 qemu-kvm
```

### 3. 比较

awk 还有比较的功能，它可以根据比较结果，返回预先设定的值。下面这几个例子中，当  $a > b$  时，即结果为真，将返回 1，否则返回 0。在比较的过程中要注意变量的类型。当进行字符串比较时，2 是大于 10 的。因为 2 的 ASCII 码大于 1 的 ASCII 码，这就是逐位比较产生的结果。

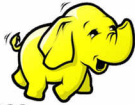
```
[root@station103 ~]# awk 'BEGIN{a=2;b=10;print (a>b)?1:0}'
0
[root@station103 ~]# awk 'BEGIN{a="2";b="10";print (a>b)?1:0}'
1
[root@station103 ~]# awk 'BEGIN{a="2";b="1";print (a>b)?1:0}'
1
```

### 4. 求和与求平均值

在当前的用户家目录中存在以下这些对象，其中第五列是 Size，现在我们要求取这些对象的 Size 总和与平均值。

```
[root@station103 ~]# ll
total 72
-rw-r--r--  1 root root    13 Sep 20 17:57 1
-r-xr-xr-x  1 root root 4096 Aug 22 17:06 chroot
-rw-r--r--  1 root root    60 Sep 21 15:56 data
-rw-r--r--  1 root root    69 Sep 17 18:50 eval.sh
-rw-r--r--  1 root root   420 Sep 19 11:03 getopt1.sh
-rw-r--r--  1 root root   457 Sep 19 17:59 getopt2.sh
-rw-r--r--  1 root root     0 Sep 19 14:36 getopt-parse.bash
-rw-r--r--  1 root root   272 Sep 18 16:53 menu.sh
-rwxr-xr-x  1 root root 20480 Aug  7 17:52 merge
-rw-r--r--  1 root root   748 Sep 15 17:35 null.sh
-rw-r--r--  1 root root    27 Sep 17 17:39 quotes.sh
-rw-r--r--  1 root root   200 Sep 15 16:31 return.sh
-rw-r--r--  1 root root    98 Sep 15 17:59 shift.sh
-rw-----  1 root root   100 Sep 20 16:04 text
-rwxr-xr-x  1 root root 4096 Sep 19 17:06 tmp
```

这一次有点复杂，我们要将 awk 分成两部分，先计算数据，再打印结果。首先，要



过滤掉 total 这一行，叹号在这里代表逻辑非。其次，求和需要累加，求平均需要知道累加的次数，这里我们使用运算符 += 来实现。这两个值都有着落了，最后我们就可以使用 printf 语句进行结果输出了。因为 printf 支持算术表达式，所以对平均值的求取也一并纳入 END 中。

```
[root@station103 ~]# ll |awk '!/total/ {sum+=$5;line+=1} END {printf "Sum = %d\nLines = %d\nAverage = %d\n", sum,line,sum/line}'
Sum = 31136
Lines = 15
Average = 2075
```

## 5. 求最值

理解了求和与求平均值之后，接着上面这个例子，我们再来学习一下如何求取最值。我看到网上有很多求最值的 awk 示例采用了 BEGIN 初始化的思路。首先定义一个初始值变量 N，然后用取值结果和 N 做比较，符合条件的就替换。求最大值时将 N 设成 0，求最小值时将 N 设置成一个极大数，比如  $2^{1024}$ 。其实这种做法并不科学，因为取值结果是未知的。假如你取到的值都是负数，那最大值就变成 0 了。这样一来，结果就错了。

其实求解最值的思路很简单，先将第一个数赋值给 N，然后再逐一比较即可。不要事先在 BEGIN 里定义 N，一旦定义就要赋值，但此时还未读取到数据，这就和赋值之间产生了矛盾。

我们完全可以直接引用一个不存在的变量。而此时它的值为空，借助条件语句 if 就能实现初始化的功能。这样的方法是不是更加巧妙呢？

```
// 求取最小值
[root@station103 ~]# ll|awk '!/total/ {if(min=="") min=$5; if($5<min) min=$5}
END{print min}'
0
// 求取最大值
[root@station103 ~]# ll|awk '!/total/ {if(max=="") max=$5; if($5>max) max=$5}
END{print max}'
20480
```

前面我们讲过，比较时要注意变量的类型。我们这里比较的是 Size，所以必须是一个数字。网上有很多争议，说求最值变量一定要加零，这样可强制将其转换成数字。但在这个例子中，我们没有这样做，结果也是对的。并未复现出网络上的反例，笔者不太确定这是否和系统版本的改进有关。至于到底要不要加零，关键还是取决于变量类型。加零是一种保险的做法，如果不放心，使用一下也无可厚非。

下面这个例子展示了 awk 进行最值求取的全过程。我们在这里面还使用了替换函数，将原来的最大值变成了负数。所以，最终结果也发生了变化。

```
[root@station103 ~]# ll|awk '!/total/ {sub("20480","-20480"); if(max=="") max=$5;
if($5>max) max=$5; print$5"\t"max} END{print max}'
```

```
13      13
4096    4096
60      4096
69      4096
420     4096
457     4096
0        4096
272     4096
-20480   4096
748     4096
27       4096
200     4096
98       4096
100     4096
4096    4096
4096
```

## 6. 替换

`sub()` 是替换函数，我们可以在 `awk` 内部实现 `sed` 的功能。下面这个例子实现了打印磁盘使用率大于 10% 的文件系统。选项 `-P` 表示输出结果遵守 POSIX 标准，一行一个条目。如果不加选项 `-P`，则信息过长会折行显示，导致 `awk` 在取值时出错。

```
[root@station103 ~]# df -Ph
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2        50G   6.4G   41G   14% /
tmpfs           32G    8.0K   32G    1% /dev/shm
/dev/sda1       194M    29M   155M   16% /boot
/dev/sda5      1018G    27G   940G    3% /export

[root@station103 ~]# df -Ph|awk '!/Filesystem/ {sub("%","",$5); if($5>10) print}'
/dev/sda2 50G 6.4G 41G 14 /
/dev/sda1 194M 29M 155M 16 /boot
```

当然这个替换看上去不太优雅，因为百分号被删除了。我们可以用加零强制转换的方式做些改进。

```
[root@station103 ~]# df -Ph|awk '!/Filesystem/ {if($5+0>10) print}'
/dev/sda2        50G   6.4G   41G   14% /
/dev/sda1       194M    29M   155M   16% /boot
```

## 7. 参数传递

如果你想将外部的一个值当作参数传递进来，可以使用选项 `-v` 自定义一个 `awk` 的内部变量。下面这个示例用于匹配特定 PID 的行，然后提取进程名称。由于匹配项是变量，而这个变量值是来自于外部的，所以要用 `-v` 来传递参数。另外，匹配时不能采用 `/var/` 这种形式，这样 `awk` 会把 `var` 当作一个字符串来匹配。正确的用法是 `~`，它会读取变量的值，然后对读到的值进行匹配。



```
[root@station103 ~]# cat -n awk.sh
```

```
1  #!/bin/bash
```

```
2
```

```
3  process=`ps -p $1|awk -v var=$1 '$0 ~var {print $NF}'`
```

```
4  echo $process
```

// 执行结果如下

```
[root@station103 ~]# sh awk.sh 1
```

```
init
```

## 15.3 字符处理

如果说文本处理是剔骨削肉,那么字符处理就是剥茧抽丝了。虽然我们能用 `awk` 从文本里抓取特定的字段,但在某些时候,这些原始内容并不一定是程序想要的。所以,字符处理是更加细粒度的操作。

### 15.3.1 字符的转义

在 Shell 里,很多字符并非只是一个普通的文本符号。例如下面这些字符,它们本身还具有一些特殊的含义。

- ❑ `~`——代表用户家目录。
- ❑ `{}/[]/()`——在 Shell 操作时,具有合并同类项、标识界限和修改执行顺序的特殊功能。
- ❑ `>/>>`——输出重定向。
- ❑ `</<<`——输入重定向。
- ❑ `&`——将命令置于后台运行。
- ❑ `&&`——与连接符,用于连接前后两条命令。当前面的命令返回成功,后面的命令才会执行。
- ❑ `||`——或连接符,用于连接前后两条命令。当前面的命令返回失败,后面的命令才会执行。
- ❑ `;`——无条件连接符,用于连接前后两条命令。两条命令会按照顺序无条件地相继执行。
- ❑ `|`——管道符,用来连接前后两条命令。前一条命令的输出结果将作为后一条命令的输入参数来使用。
- ❑ `$`——美元符,用于求取变量的值。
- ❑ ```——反引号是命令替换符,反引号内的字符会被 Shell 当作命令来执行,并返回执行结果。
- ❑ `!`——叹号是 `histroy` 替换符,叹号后面加数字代表执行第几条历史命令,双叹号则代表执行最后一条历史命令。

## 1. 转义符

既然这么多字符都带有特殊含义，如果要把它们用作普通文本，又该怎么办呢？这时，我们就要用到另外一种特殊的字符——转义符。

转义符的英文是 Escape Character。它使用 Backslash，也就是我们常说的反斜线。位于转义符后面的单字符会被转义，该字符的特殊含义会被消除。就像下面这个例子，美元符被转义后，失去获取变量值的能力，此时的它就是一个普普通通的美元符号而已。

```
[root@station103 ~]# echo Your cost: \$5.00
Your cost: $5.00
```

## 2. 引号

引号也有转义功能。引号分为两类——双引号和单引号。双引号可以转义除美元符、反引号、反斜线和叹号以外的所有字符。而单引号则能够转义除它自己以外的所有字符。

我们来看下面这个例子。系统内置的环境变量 PWD 用来打印当前的工作目录。在添加双引号后，执行结果没有发生改变，这说明双引号尊重美元符的取值行为。在修改成单引号后，输出结果变成了 \$PWD。

```
[root@station103 ~]# echo $PWD
/root
[root@station103 ~]# echo "$PWD"
/root
[root@station103 ~]# echo '$PWD'
$PWD
```

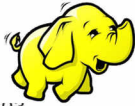
由于引号的语法歧义，单引号无法实现自我转义。请看下面这个例子，我们本来是要用最外面的那组单引号做转义的。但不幸的是，第一个用来转义的单引号与第二个需要被转义的单引号组成了一对，后面两个也是同理。最终的结果就是，单引号相互之间被抵消掉了。

```
[root@station103 ~]# echo '$PWD'
/root
```

转义单引号可以使用转义符或者双引号来完成。有趣的是，当双引号中包含单引号时，双引号也是采用转义符来转义单引号的。

```
[root@station103 ~]# echo \"'$PWD'\"
'/root'
[root@station103 ~]# echo "'$PWD'"
'/root'
// 将第二条命令保存到文件 quotes.sh，然后使用 -x 选项观察转义过程。
[root@station103 ~]# sh -x quotes.sh
+ echo "'\' '/root\''"
'/root'
```

请注意，由于字体显示的原因，两种引号挤在一起，相互之间难以区分。所以笔者才



特意在这里加了空格，方便读者看清楚。命令在实际执行过程中，引号之间是没有空格的。另外，最后一个示例中的输出结果全是单引号，它们之间也没有空格。这里多出来的单引号是扩展信息的部分，用于标记字符串边界，不作为转义使用，也不存在语法歧义的问题。

### 15.3.2 字符串截取

如果字符串是冗余的，我们只想保留其中一部分内容，这就需要使用截取操作来完成。字符串的截取大致分为两类，一种方式是用 `echo` 截取变量中的字符串，另一种方式是用 `expr` 直接截取一个字符串。

#### 1. 按照变量中的字符位置截取

```
# echo ${VARIABLE:[Z±]X:Y}
```

`X` 是指起始字符的位置。如果 `X` 是正数，表示从左边第 `X+1` 个字符起向右侧截取，负数则表示从右向左截取 `X` 个字符。注意：负数不能直接表示，需要使用一个算术表达式 `Z-X` 来替代，通常情况下 `Z` 等于 0。

算术表达式 `Z±X` 的结果将作为 `X` 的新值来使用。所以，`0-3` 和 `2-5` 的结果是一样的，加法也是同理。我们建议，尽量不要使用多余的表达式，以增加代码的可读性。正数直接使用 `X`，负数使用 `0-X`，这才是最佳的表达方式。

```
// 首先给变量赋值
[root@station103 ~]# str=abcdefg
// 算术表达的结果决定了截取方式
[root@station103 ~]# echo ${str:0-3} ${str:2-5}
efg efg
[root@station103 ~]# echo ${str:10-5} ${str:2+3} ${str:5}
fg fg fg
[root@station103 ~]# echo ${str:10-8} ${str:1+1} ${str:2}
cdefg cdefg cdefg
```

`Y` 会对截取后的字符串做二次裁剪，它将从左向右保留 `Y` 个字符。就像下面这两个例子所展示的那样。

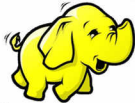
```
[root@station103 ~]# echo ${str:0} ${str:0:2}
abcdefg ab
[root@station103 ~]# echo ${str:0-5} ${str:0-5:2}
cdefg cd
```

#### 2. 匹配变量中的关键字截取

```
# echo ${STRING#[#]*CHAR}
# echo ${STRING%[%]CHAR*}
```

第一条语法用于左截断，查找关键字的方式是从左向右，`#` 表示匹配第一个关键字，`##` 表示匹配到最后一个关键字。左截断将连同关键字及其左侧的所有内容一并删除。





第二条语法用于右截断，查找关键字的方式是从右向左，% 表示匹配第一个关键字，%% 表示匹配到最后一个关键字。右截断将连同关键字及其右侧的所有内容一并删除。

我们来看下面这个例子。首先给变量 url 赋值，字符串是一个 URL 链接。我们以斜线作为关键字，分别观察左截断和右截断的效果。

```
// 首先给变量赋值
[root@station103 ~]# url=http://example.com/pub/index.html

// 左截断效果
[root@station103 ~]# echo -e $url"\n"${url#*/*}"\n"${url##*/}
http://example.com/pub/index.html
/example.com/pub/index.html
index.html

// 右截断效果
[root@station103 ~]# echo -e $url"\n"${url%/*}"\n"${url%%/*}
http://example.com/pub/index.html
http://example.com/pub
http:
```

### 3. 使用 expr 命令截取

```
# expr substr STRING X Y
```

expr 命令不但可以运算，它还有截取字符串的功能。使用方法也很简单，从左边第 X 个字符起截取 Y 个字符。就像下面这个例子所示的那样。

```
[root@station103 ~]# expr substr "example.com" 9 3
com
```

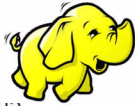
## 15.4 数组

好不容易完成了文本和字符的处理工作，我们要把劳动成果保存起来，以备程序随时调用。放到变量里面是个不错的选择，但是一个变量只能保存一个数值。有时候我们需要存储一组数据，而数据的个数又不确定。此时，变量就无法胜任了。很多初学者喜欢用临时文件来存储，比如用 tee 命令去生成，这可不怎么高明。我好不容易刚把数据从文件里取出来，还没怎么用呢，就要再写回到磁盘中，这将引起大量低效的磁盘 I/O。

现在，就要用到数组这个概念了。数组是一种存储结构，它能够存储多个数据。数组分两类，一种称为索引数组，存储对象是一个值；另外一种称为关联数组，存储对象是一组键值对。

### 1. 索引数组

由于索引数组只存储数值，所以它的数据源应当是唯一的，相当于一个变量具有多个值。举个例子，我要获取当前系统的 DNS Server，可以执行如下这条命令。



```
[root@station103 ~]# awk '/^nameserver/ {print $NF}' /etc/resolv.conf
202.106.0.20
8.8.8.8
```

我们发现 DNS Server 的地址并不唯一，但它们都属于 DNS Server 的值。这就是一个典型的索引数组的关系。我们使用一个简单的 for 循环就能实现数据的存储。索引数组是有序的，新增对象从数组的末尾依次追加。

```
[root@station103 ~]# for i in `awk '/^nameserver/ {print $NF}' /etc/resolv.conf`
> do
>   array+=($i)
> done
```

索引数组用下标表示数据在数组中的存储位置，并用方括号包裹。下标从 0 开始，表示第一个数据。@ 或 \* 则表示数组中所有的数据。此外，对于数组的操作还有以下两种用法。

□ #——获取长度；

□ !——获取下标。

```
[root@station103 ~]# echo ${array[@]}
202.106.0.20 8.8.8.8
[root@station103 ~]# echo ${array[0]}
202.106.0.20
[root@station103 ~]# echo ${array[1]}
8.8.8.8
```

// 获取 array[0] 的长度

```
[root@station103 ~]# echo ${#array[0]}
12
```

// 获取存储数据的个数

```
[root@station103 ~]# echo ${#array[@]}
2
```

// 获取数组内的下标

```
[root@station103 ~]# echo ${!array[@]}
0 1
```

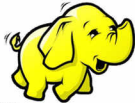
## 2. 关联数组

与索引数组不同，关联数组存储的是一个键值对，即 Key-Value。这种存储结构很适合配置文件的读取，最典型的就是网卡的配置文件。

```
[root@station103 ~]# egrep '^IPADDR|^NETMASK' /etc/sysconfig/network-scripts/ifcfg-lo
IPADDR=127.0.0.1
NETMASK=255.0.0.0
```

引入一个数组需要事先用 declare 命令声明。-A 表示关联数组，-a 表示索引数组。前面使用索引数组时，我们并没有提前声明，这是因为数组的默认类型为索引数组。由于键值是成对映射的关系，所以关联数组是无序的，新增对象将以乱序的形式插入数组。

// 要提前声明关联数组才可以存储 Key-Value



```
[root@station103 ~]# declare -A dict
[root@station103 ~]# for i in `egrep '^IPADDR|^NETMASK' /etc/sysconfig/network-
scripts/ifcfg-lo`
> do
>   key=`echo $i|cut -d= -f1`
>   value=`echo $i|cut -d= -f2`
>   dict+=([$key]=$value)
> done
```

关联数组的用法和索引数组类似。它没有下标的概念，取而代之的是键，用方括号包裹。@ 或 \* 则表示数组中所有的值，! 用来获取键名。

```
// 读取关联数组中存储的所有 Key-Value
[root@station103 ~]# for i in `echo ${!dict[@]}`
> do
>   echo -e $i"\t"${dict[$i]}
> done
NETMASK 255.0.0.0
IPADDR 127.0.0.1
```

### 3. 数组的删除

两种数组相互之间是不能转换的，如果你在定义一个数组时遇到如下的错误提示，说明这个数组之前被定义成了另外一种类型。使用时需要先删除它之前的状态。

```
[root@station103 ~]# declare -A array
-bash: declare: array: cannot convert indexed to associative array
```

删除数组或者数组内的值要用到 `unset` 命令。

```
# unset array
# unset array[1]
# unset dict[IPADDR]
```

## 15.5 算来算去

计算是信息处理中另外一个重要的部分。简单的四则运算可以用 `$()`、`expr`、`bc` 等多种方式来实现，本节我们主要介绍计算当中的一些高级用法。

### 15.5.1 比较

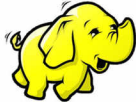
`expr` 支持形式简洁的表达式比较，比较的对象可以是算术表达式、数字或字符串等。

#### (1) 等式比较

表达式为真返回 1，为假返回 0。

```
[root@station103 ~]# expr 2 '<=' 23
1
[root@station103 ~]# expr 2 '>=' 23
```





```
0
[root@station103 ~]# expr 2 '!=' 23
1
[root@station103 ~]# expr 2 '=' 23
0
```

### (2) 逻辑或比较

如果第一个表达式为 0 或空则返回后者，否则返回前者。

```
[root@station103 ~]# expr 2 '|' 23
2
[root@station103 ~]# expr 0 '|' 23
23
[root@station103 ~]# expr '' '|' 23
23
```

### (3) 逻辑与比较

如果任意一个表达式为 0 或空则返回 0，否则返回前者。

```
[root@station103 ~]# expr 2 '&' 23
2
[root@station103 ~]# expr 0 '&' 23
0
[root@station103 ~]# expr 23 '&' ''
0
```

## 15.5.2 字符串计算

字符串计算的典型案例是密码设定。安全策略不允许使用长度不足的密码。expr 正好能够胜任这项有趣的工作，它支持以下的这些函数用法。

### (1) 计算给定字符串的长度

length 方法用于计算一个指定字符串的长度。

```
[root@station103 ~]# expr length "example.com"
11
```

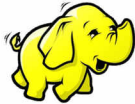
### (2) 计算给定字符的位置

index 方法可根据给定字符计算其位于指定字符串的第几位。匹配顺序是从左向右进行的。注意，当给定多个字符时，它会挑最先匹配的那一个。

```
[root@station103 ~]# expr index "example.com" m
4
[root@station103 ~]# expr index "example.com" mx
2
```

### (3) 计算匹配字符的个数

match 方法可根据给定的表达式计算匹配字符的个数。注意匹配是从字符串的第一个字符开始的，如果第一个字符匹配失败则返回 0。



```
[root@station103 ~]# expr match "example.com" e.*e
7
[root@station103 ~]# expr match "example.com" a.*e
0
```

### 15.5.3 精度与长度

`$()` 和 `expr` 都只适用于整数计算，对计算结果也是取整处理。当涉及浮点数运算时，需要用到 `bc` 这个强大的计算工具，它能胜任更加复杂的计算任务。

从下面这个例子中能够看到，默认情况下 `bc` 对于除法的计算结果也做了取整处理。为了得到精确的数值，需要事先用 `scale` 定义浮点数的精度。

```
[root@station103 ~]# echo '1/3'|bc
0
[root@station103 ~]# echo 'scale=3; 1/3'|bc
.333
```

当精度定义完成后，再次执行得到的结果是正确的。但是我们发现了一个问题，个位上缺了个 0。尽管这并不影响 `bc` 后续的计算，但看起来还是别扭。`bc` 和 `awk` 一样，它也拥有强大的流程控制功能。我们使用条件语句 `if` 可以实现补零的操作。

下面这个示例语句的含义就是——先将结果赋值给一个变量 `a`；假如 `a<0`，在 `a` 的前面先打印一个 0，实现了补零的操作；最后还要加一个换行符，这是因为 `print` 语句默认是不换行的。

```
// 当 a 小于 0 就补位一个 0
[root@station103 ~]# echo 'scale=3; a=1/3; if (a<1) print 0,a,"\n" else print
a,"\n"|bc
0.333
// 当 a 不小于 0 还是正常输出
[root@station103 ~]# echo 'scale=3; a=10/3; if (a<1) print 0,a,"\n" else print
a,"\n"|bc
3.333
```

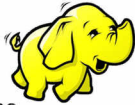
`scale()` 和 `length()` 这两个函数用于求取表达式结果的精度与长度。

```
// 计算 a+b 的结果，即 10.001，该数值精度是 3 位小数，数值的长度为 5。
[root@station103 ~]# a=0.001; b=10
[root@station103 ~]# echo "c=$a+$b; scale(c); length(c)" |bc
3
5
```

### 15.5.4 进制转换

进制转换是计算时经常碰到的需求。`bc` 提供了强大的进制转换功能，其中 `ibase` 表示输入侧的进制类型，`obase` 表示输出侧的进制类型。如果没有找到 `obase`，默认将以十进制的方式输出计算结果。

```
[root@station103 ~]# echo "ibase=8;101" |bc
```



```
65
[root@station103 ~]# echo "ibase=16;B5-A3" |bc
18
[root@station103 ~]# echo "ibase=16;obase=2;B5-A3" |bc
10010
```

这里要特别注意一点，如果 `ibase` 和 `obase` 同时要指定，`obase` 的数字表达将跟从 `ibase` 的设置。请看下面这个例子，我们要将十六进制的  $100_{(16)}$  转换成十进制的数字，`obase` 这里不能写 10。因为现在是十六进制的表达方式，这里的 10 表示的是  $10_{(16)}$ ，转换成十进制后还是 16。也就是说，输入输出都是十六进制，根本就没有做任何转换。正确的写法应当是 0A 才对。

```
[root@station103 ~]# echo "ibase=16;obase=10;100" |bc
100
[root@station103 ~]# echo "ibase=16;obase=0A;100" |bc
256
[root@station103 ~]# echo "ibase=16;100" |bc
256
```

## 15.6 表面文章

当程序完成了对数据的处理，最后一步就是结果的输出。虽然输出是三大环节中最简单的一个，但它也是最重要的。由于输出结果要面向用户，假使它在排版及可读性上的表现非常糟糕，让用户难以理解或者厌烦，即使前面的工作做得再好，这样的程序也是失败的。

使用换行、分隔线与制表符是实现版式优化的基础。例如，类似 Ansible 或者 SaltStack 这种执行结果的返回，产生的是多组信息集合。组间可以用减号组成的单横线分隔，表头信息可以考虑用等号构成的双横线来标记。

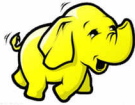
像 `vmstat` 这种表结构数据的展示，列间分隔请使用制表符，让数据在垂直方向上也是对齐的。这一点对于类型相同的数据尤为重要。不然用户在查看数据时，很容易将其对应到了错误的表头上，也就是咱们常说的“看串行了”。总而言之，千万不要吝惜你的分隔符，美化输出也是一门学问。

一方面，我们要不断修饰美化我们的输出结果；另一方面，也要注意信息泛洪带来的拖累。我们一直在说，信息过量会分散用户的注意力，并产生阅读疲劳。信息是给人看的，只要展示最关键的部分就可以。那什么是最关键的呢？你只要输出用户关心的内容就好，其他部分可以保存到日志里面，或者作为调试级信息输出。

此外，颜色显示能有效地吸引用户的注意力。System V 类的服务在启停时会用颜色告知用户操作执行的结果。在我们自己的程序中，一样可以引入这种做法。

这项功能是 `echo` 命令的扩展，它的语法格式如下所示。注意这里的花括号只是为了标明变量而已。实际写的时候，不需要花括号的参与。





```
echo -e "\033[XY;Xm{STRING}\033[{CONTROL}]"
```

\033 是转义的意思，我们也可以用 `\e` 来代替。XY 是由两个数字组成的，其中 X 用于描述图层，Y 用于描述颜色。XY 可以写一组或者是两组，使用两组 XY 意味着同时控制前景色和背景色，这两组数字之间需要用分号隔开。

1) 图层分为两种：假如 X=3，代表前景色，则 Y 描述的就是字体颜色；假如 X=4，代表背景色，则 Y 描述的就是屏幕颜色。

2) 颜色共分为 8 种，即 0——黑色，1——红色，2——绿色，3——黄色，4——蓝色；5——紫色；6——天蓝，7——白。

STRING 就是我们的消息正文，CONTROL 表示控制字符。前面设置颜色的 XY 也是一种控制字符，控制字符最后都有一个字母 m 作标记。当我们自定义了颜色以后，终端后面的所有输出都要使用这个颜色。所以，我们在标记完特定的字符串后，需要恢复终端原有的状态，否则就乱套了。这里用了一个 0m 代表关闭掉 echo 的所有属性。此外，echo 还支持以下这些控制字符。

```
1m: 设置高亮度    4m: 下划线    5m: 闪烁    7m: 反显    8m: 消隐
nA: 光标上移 n 行  nB: 光标下移 n 行  nC: 光标右移 n 行  nD: 光标左移 n 行
y;xH: 设置光标位置  2J: 清屏    K: 清除从光标到行尾的内容
s: 保存光标位置    u: 恢复光标位置    ?25l: 隐藏光标    ?25h: 显示光标
```

我们举两个例子。首先模拟 System V 服务的执行结果的返回。红色字体代表失败，绿色字体代表成功。这里我们稍微做了一点优化，就是增加了高亮显示。刚才已经讲过，颜色设定也是一种控制字符，如果要使用多组控制字符，需要用分号分隔。这个例子中展示的就是前景色和高亮显示的一种组合。大家完全可以根据自己的想象去设计，不必局限于只在颜色上做文章。

```
# echo -e "\033[31;1mFAILED\033[0m"
# echo -e "\033[32;1mOK\033[0m"
```

我们再来看另外一个例子。首先给高亮背景绿的 Success 定义别名 vv，然后用“与连接符”把 vv 放在待执行命令的后面。因为 Success 非常显眼，如果我们能看到它，就证明前面命令的执行是成功的，就不必再查看返回值了。

```
[root@station103 build]# alias vv='echo -e "\033[42;1m Success! \033[m"'
[root@station103 build]# ls && vv
build_base_gcc43-64bit.0000 build_base_mac-example.0000 list
Success!
```

## 15.7 典型案例

本章的最后，我再给大家分享一些实际工作中用到的案例。这些案例中既有对前面知识点的总结，也有综合应用的展现。

## 1. IP 地址的排序

排序是很常见的需求，我们举一个经典案例，看看如何实现对 IP 地址的排序。首先，我们创建一个带有 IP 地址列表的文件。

```
[root@station103 ~]# cat ip
10.1.2.3
192.168.9.1
192.168.10.1
192.168.11.1
192.168.20.1
202.106.0.20
8.8.8.8
172.16.1.253
```

直接使用 `sort` 之后并没有达到我们预期的效果，`sort` 只是对字符的 ASCII 码进行了排序，这不是我们想要的。

```
[root@station103 ~]# cat ip |sort
10.1.2.3
172.16.1.253
192.168.10.1
192.168.11.1
192.168.20.1
192.168.9.1
202.106.0.20
8.8.8.8
```

接下来我们使用选项 `-n`，按照数字进行比较，但结果仍不理想。很显然，192.168.9.1 这个 IP 地址被排在了 192.168.20.1 的后面。这是因为 `sort` 默认是按照字符顺序来比较的。由于同处字符串的第九位，且 9 大于 2，所以才产生了这种错误的结果。

```
[root@station103 ~]# cat ip |sort -n
8.8.8.8
10.1.2.3
172.16.1.253
192.168.10.1
192.168.11.1
192.168.20.1
192.168.9.1
202.106.0.20
```

IP 地址的排序需要根据字段来操作。选项 `-t` 用于设置分隔符，选项 `-k` 用于指定范围。逗号前面的数字代表起始列，后面的数字代表结束列。我们使用句点作为分隔符，每次排序的范围都只限定在当前的字段内。这样一来，在第三字段排序时，就变成了 9 与 20 的比较，最终结果也就符合了我们的预期。

```
[root@station103 ~]# cat ip |sort -n -t. -k1,1 -k2,2 -k3,3 -k4,4
8.8.8.8
```

```

10.1.1.2.3
172.16.1.253
192.168.9.1
192.168.10.1
192.168.11.1
192.168.20.1
202.106.0.20

```

## 2. 取网卡的 IPv4 地址

说完了 IP 地址的排序，IP 地址的获取也是我们经常要用到的。但 `ifconfig` 和 `ip` 命令的信息输出不是特别友好。我们可以用下面这段代码来实现网卡设备名称和 IPv4 地址的获取。

```

[root@station103 ~]# cat -n dev.sh
 1  #!/bin/bash
 2
 3  echo `ip addr show $1|awk -v var=$1 '/inet / {print var":"$2}`

[root@station103 ~]# sh dev.sh lo
lo:127.0.0.1/8

```

## 3. 单列数据的动态输出

这是笔者以前做巡检时遇到的一个需求——如何持续地动态观测可用内存的变化？因为服务器的内存配置比较高，`free` 值的默认单位是 KiB，需要我们将它转换成 GiB。

```

[root@station103 ~]# vmstat 1
procs -----memory----- ---swap-- -----io----- --system-- -----cpu-----
 r  b   swpd free   buff   cache     si   so    bi   bo    in     cs us sy   id wa st
 0  0   3684 993168 23611864 11815636    0    0     0    9    0      0 0 0 100 0 0
 0  0   3684 993036 23611864 11815636    0    0     0  40 2460  5501  0 0 100 0 0
 0  0   3684 988272 23611864 11815636    0    0     0 340 3149  6748  0 0 100 0 0
^C

```

列输出自然要用到 `awk` 命令。至于单位转换，`vmstat` 有选项 `-S` 可以使用，不过它不支持 GiB。既然如此，我们直接用 `awk` 来实现好了。这里的 `%.2f` 代表保留保留两位小数的 `float` 数据类型。

```

[root@station103 ~]# vmstat 1|awk '/[0-9]/ {printf("free: %.2f GB\n",
    $4/1024/1024)}'
free: 0.90 GB
free: 0.91 GB
free: 0.91 GB
^C

```

## 4. 多行合并

多行合并的问题常将一些初学者难倒。我们为大家提供了三种解决方案。示例文件我们还是使用之前的 `text`。



```
[root@station103 ~]# cat -n text
```

```
1 bird work good seak
2 mark bird bird rest
3 push deep bird bush
4 desk jeep need year
5 beat meet seat bird
```

### (1) sed

解法如下：

```
sed ':a;N;s/\n/ /;ta' text
```

由于 sed 是按行处理的，所以它一次只能读取一行。而合并工作必须要读取两行才可以，这里的 N 就是让 sed 读取下面一行的意思。接着将换行符替换成空格完成合并。就像下面这样，两两合并，五行变成了三行。

```
[root@station103 ~]# sed 'N;s/\n/ /;t' text |cat -n
```

```
1 bird work good seak mark bird bird rest
2 push deep bird bush desk jeep need year
3 beat meet seat bird
```

光两两合并还不行，我们要把所有行合并到一行里。这要用到跳转命令，它很像 goto 语句，让 sed 回到某一个位置。这个位置是哪里呢？我们在最前面设置了一个名为 a 的 Label，用来表示文件的开头。ta 代表当前行处理完成后跳回到 a 的位置，也就是回到文件的开头。然后 sed 会重新开始新一轮的合并，直到合并完所有的行为止。

```
[root@station103 ~]# sed ':a;N;s/\n/ /;ta' text |cat -n
```

```
1 bird work good seak mark bird bird rest push deep bird bush desk jeep
  need year beat meet seat bird
```

### (2) awk

解法如下：

```
awk 'BEGIN{RS=EOF}{gsub(/\n/, " ");print}' text
```

RS 全称是 Record Separator，即记录分隔符，它是 awk 的内置变量。RS 默认为 \n，也就是说 awk 每次只处理一行内容。现在我们将 RS 改成 EOF，EOF 代表文件结束。此时对于 awk 来讲，整个文件就是一行内容。它会一次性把文件全部读到内存当中处理。gsub() 函数就是用来删除换行符的。

```
[root@station103 ~]# awk 'BEGIN{RS=EOF}{gsub(/\n/, " ");print}' text |cat -n
```

```
1 bird work good seak mark bird bird rest push deep bird bush desk jeep
  need year beat meet seat bird
```

### (3) xargs

解法如下：

```
cat text |xargs
```

`xargs` 用于将标准输入转换成命令行参数执行，也能实现文本格式的转化。

```
[root@station103 ~]# cat text |xargs |cat -n
1 bird work good seak mark bird bird rest push deep bird bush desk jeep
need year beat meet seat bird
```

既然三种方法都能实现，那么它们的执行效率又如何呢？我们做个实验来论证一下。首先创建一个十万行的文本文件，然后分别用 `time` 统计三个方案的执行时间。注意：我们要把输出结果重定向给 `null`。不让他输出到屏幕的原因有两个：第一文本内容过多，我们没必要看它；第二 `dump` 到屏幕的时间过长，而且会被算到等待时间里。但这和程序执行效率无关，反而干扰结果的判断。

```
// 创建样本文件
[root@station101 ~]# for i in $(seq 1 100000); do echo $i; done > lines

//sed 的执行时间
[root@station101 ~]# time (sed ':a;N;s/\n//;ta;' lines > /dev/null)

real    0m23.296s
user    0m23.300s
sys     0m0.000s

//awk 的执行时间
[root@station101 ~]# time (awk BEGIN{RS=EOF}'{gsub(/\n/, " ");print}' lines > /dev/null)

real    0m0.017s
user    0m0.010s
sys     0m0.000s

//xargs 的执行时间
[root@station101 ~]# time (cat lines|xargs > /dev/null)

real    0m0.015s
user    0m0.010s
sys     0m0.000s
```

因为 `sed` 是逐行处理的，所以它的执行效率最差。而后两种方案都是将内容一次性读取完，所以它们的处理速度要快得多。

既然提到了多行合并，我们顺便再说一说多文件合并的方法。这是一个真实的业务需求：业务方有大量的文本文件需要做合并，单个文件不大，但是文件数量非常多。开始他们采用了文本追加的方式，一连执行了几个小时都没完成。那是自然的，因为时间全都浪费在 I/O 读写上了。只要你的内存足够大，将所有文件读取到内存，然后再一次性写入是最好的方式。

```
# cat * |xargs > file
```



## 5. trap

信号是一种原始的进程通信机制，几乎每一个用户都使用过它。信号传递通过 kill 命令来实现，我们经常使用 SIGKILL 来强行杀死一个运行中的进程。除了 SIGKILL 以外，按 Ctrl+C 组合键中断操作也是一个典型的信号实例，它被称作 SIGINT。中断操作看上去很正常，用户出于某种原因想要放弃当前的执行，但在很多场合下中断是有害的。比如，用户在程序编译或者修改文件的过程中发出了一个中断操作，如果没有妥善的回滚策略，数据在中断后就会残存很多中间态的内容。再次执行时，就有可能会出现问题——文件已存在或者数据找不到等。

trap 命令用来捕获一个特定的信号，当该信号出现时，trap 将执行一条事先设定好的命令。你可以把 trap 当做针对某个特定信号的异常处理，它甚至还可以禁用原先默认的信号行为。它的语法格式如下所示。

```
trap [OPTION] [Command] [Signal]
```

我们就以 SIGINT 来举例说明。下面这个语句表明，当系统触发信号 SIGINT 时，就在屏幕上打印 Ctrl-C Pressed! 这段内容。而且它会一直存留在这个 Session 环境中等待 SIGINT 的到来，你按几次 Ctrl+C 组合键，它就会打印几次，直到退出这个 Shell 为止。

```
[root@station103 ~]# trap "echo Ctrl-C Pressed\!" 2
[root@station103 ~]# ^CCtrl-C Pressed!

[root@station103 ~]# ^CCtrl-C Pressed!

[root@station103 ~]#
```

下面这个例子中，命令部分是空的，它的意思是禁止 SIGINT。这条指令执行后，再按 Ctrl+C 组合键将不起任何作用。

```
[root@station103 ~]# trap '' 2
```

trap 可以捕获的信号种类很多。关于信号的描述，既可以是数字，也可以是信号名称。两者之间的映射关系可以执行 trap -l 来查询。关于每个信号的具体含义可以使用 man 7 signal 来查询。

```
[root@station103 ~]# trap -l |head -1
1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
```

如果想要了解当前 Shell 中设置了哪些 trap，可以执行 trap -p 来查询。如果要移除某一个 trap，可以使用如下方式恢复信号的默认操作。如果当前 Shell 关闭，trap 也会随之结束。

```
[root@station103 ~]# trap -p
trap -- 'echo Ctrl-C Pressed\!' SIGINT
[root@station103 ~]# trap 2
[root@station103 ~]# trap -p
```



```
[root@station103 ~]#
```

## 6. 全局变量的定义

假如我们有一些自定义的全局变量，请将它们单独置于一个脚本中（假设该脚本名为 global.sh），然后在那些需要全局变量的脚本中添加如下这行。

```
source /PATH/global.sh
```

注意：source 这个全局变量脚本时，一定要确认检查脚本的位置。本例中没有做检查是为了简化不必要的过程。实际应用时一定要确保执行是成功的，否则我们提取的全局变量都将为空，这肯定会影响最终的执行结果。切记切记！

```
// 全局变量脚本
[root@station103 ~]# cat global.sh
#!/bin/bash
var=global
// 调用脚本
[root@station103 ~]# cat call.sh
#!/bin/bash
source /root/global.sh
echo $var
// 执行脚本时可以获取变量 var 的值，当程序结束后变量 var 是不存在的。
[root@station103 ~]# sh call.sh
global
[root@station103 ~]# echo $var
```

另外我们看到，source 命令可以让自定义变量立即生效，但是这些自定义变量不会保存到系统中。因为 sh 在执行脚本时，单独创建了一个新的 Shell。所以，这些自定义变量在程序退出后就被清除了，不用担心它们会污染现有的系统环境。

## 15.8 本章小结

本章内容以程序处理的三部曲为主线，为读者精讲了 Shell 在各个阶段可能会遇到的核心问题。本章最大的难点在于对参数传递的充分理解和掌握，这对构建一定规模的 Shell 代码来讲非常重要。此外，还请大家重视数组和文本处理的学习。

到这里，关于技术环节的内容就为大家全部讲解完了。如果你还有耐心愿意翻开最后一个篇章的话，这里将呈现一个特别篇，讲述一个 SE 的修行之路。



## 第 16 章

# 修行之路

十年磨一剑，霜刃未曾试。  
今日把示君，谁有不平事。

当我书写到这一章的时候，已是秋去冬来，转眼间又过了一年。初冬的夜空格外深邃，偶有几颗难得一见的流星，只一闪便消失在了那一幕墨色之中。我喜欢冬季的黑夜，它的到来让我远离了一切喧嚣与烦恼。每天上下班的往返路途是漫长的，少说也要花上三四个小时。说实话，这很糟糕。在汽车上，大多数人都在利用这段时间来补觉或玩手机，而此时的我却喜欢呆呆地凝视着窗外，望着远处摇曳的昏黄灯火，渐渐地陷入了沉思。

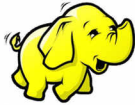
我很享受这个除了睡觉以外、唯一能够获得安宁救赎的时刻。修行之路还很长，未来该如何继续走下去，我想这是很多进入而立之年的人正在考虑的事。参加工作也已经十多年了，十年光阴转瞬即逝，很多经历耐人寻味，让我也有了一些感悟与心得。

### 16.1 系统工程师的自我修养

传统含义里，工程师是专业领域内技术精英的代名词。一名工程师应当有着丰富的理论知识，可以独立地解决技术难题，并能在实际生产中取得一定的经济价值。反观现今的IT行业，好像只要你是干技术的，都可以叫工程师，然后再根据你的技术水平去定级。说实话，这种做法非常不严谨，缺乏一个公正客观的度量标准。新员工、老员工、混什么圈子、上一家工资拿多少、跟领导的关系如何，等等，影响因素实在是太多了。不能否认，级别确实是个好东西，因为它跟工资收入挂钩啊。不过，现在的我会看得很淡。我认为一个工程师的自我修养是更加重要的东西。

斯坦尼斯拉夫斯基有一本著作叫作《演员自我修养》。一个演员好不好，取决于他的演技如何。牌子再大，出场费再多，也只能骗一次人。好演员不在于你有多大的名气或者获得了多少殊荣，观众说好才是真的好。我相信一个工程师自我修养的提升，会给他带来更





多的价值和回报。

### 16.1.1 工程师与管理员

有时候大家习惯管系统工程师叫作 SA。从个人情感上讲，我不太能接受这种叫法。这里的 A 是管理员的意思，我们内部也管 SA 叫 Support All。在整部书当中，我一直都在提 SE 这个概念，而不是 SA。在我看来，二者之间有着非常大的职业差距。

#### 1. 表和里的差别——技术深度

在技术深度上，SA 偏重于操作系统的使用，而 SE 则更注重系统内核和代码等深层次的东西。如果把操作系统比作汽车，那么 SA 就是驾驶员，而 SE 就是汽车设计师。另外，这里的技术深度是广义上的，并不局限于系统本身。因为在软件架构体系中，SE 所扮演的就是一个承上启下的角色。向上要承载业务，向下要对接基础设施，因此它的知识领域并没有明确的界线。

比方说，服务器在和网络设备进行联调时遇到了通信问题，SA 认为自己的配置是正确无误的，而 NE 那边也不承认是他的问题。如果 SA 不懂网络，NE 不懂系统，俩人又都无法提供足够的证据来证明是对方的错，那么接下来肯定就要开始推诿扯皮了。这就是我所说的技术深度不足。

#### 2. 点和面的差别——驾驭能力

驾驭能力主要是指你所能掌控的架构规模有多大。传统 IT 模型以小型机为主，规模较小，节点数量的增长也比较缓慢。而互联网模式的运维工作就不会那么简单了。

我曾经面试过一个候选人。他对薪水的要求很高。面试刚开始，他就问我待遇方面的问题。我说：“只要你有能力，薪水可以去和我的领导谈。”他说：“Linux 系统管理我全吃透了，常用服务我都会配置。”这时我问他一个问题：“假如让你一个人负责一个 3000 台服务器规模的数据中心，你能胜任吗？”他一听这话，摇了摇头说：“这也太多了吧，怎么着也得给我安排俩人，大家一块儿干啊。”于是我告诉他说：“我刚来这家公司的时候，算上我一共就俩人，三个数据中心，其中还有一个是白皮儿机房（空机房）。所有的规划设计工作都要求我独立完成，从零开始建设一个数据中心。如果依你所言，恐怕我们的工作量你是承受不了的。”

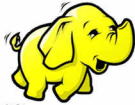
#### 3. 如何从传统运维向互联网行业转型

从传统运维向互联网行业转型，我认为需要实现两种能力上的突破。

第一个是操作效率的能力。我在面试的时候经常会问候选人，是否接触过服务器的带外管理？有的候选人会说：“这个我弄过啊，就是在 IE 浏览器里面敲一个 URL 地址，再输入用户名密码就行了。”接着我又问：“假如我有 1000 台服务器需要统计 SN，你能否以最快的速度完成呢？”如果局限于图形化的手工操作，那根本达不到互联网运维的基本







要求。

第二个是横向扩容的能力，你所能管理的规模有多大？100、10 000 还是 1 000 000？这三者之间的差距，绝对不是线性提升一百倍那么简单。要知道，量变产生质变。不同的架构都有自己的瓶颈。拿硬件监控来说，100 个节点你可以做主动监控，10 000 个节点你还能用邮件告警来扛，但是 1 000 000 个节点怎么办？告警你看得过来吗？每天数百台的故障设备，你人工报修受得了吗？

你设计的架构能支撑多久？到哪里是瓶颈？在瓶颈期之前如何平滑过渡？这些内容综合来看，就是在考验你的驾驭能力，能够承接多大规模的系统。这就是一个面的能力。

### 16.1.2 系统工程师的三颗心

《西游记》第七十九回记载了这样一段故事。

---

话说唐僧师徒一行四人行至比丘国。那比丘国王只因身染重疾，缠绵日久不愈。不想他的国丈却是个妖精，撺掇昏君要取唐僧腹中的心肝做药引子。悟空变化的假唐僧问：“和尚我可有的是心，不知你要的是哪一颗啊？”国丈言道：“我就要你的那颗黑心。”悟空道：“既如此，快取刀来。剖开胸腹，若有黑心，谨当奉命。”霎时间，腹皮剖开，竟滚落出一堆心来，但就是没有一颗黑心。

---

取经人要有拜佛求经的诚心，普度众生的佛心，悲天悯人的善心，救苦救难的慈心，矢志不渝的忠心和斩妖除魔的决心。同样，一个 SE 也要具有三颗特别的心。

#### 1. 对生产系统要怀敬畏心

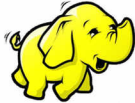
敬畏心是每一个运维人都应有的态度。像我是做互联网金融的，多少和钱都有些关系。以前有个笑话，是说某家银行系统要升级，负责升级的人在操作之前干的第一件事，就是先把自己在这家银行里存的钱全都给取了出来。

曾经有个网友和我争辩，说误操作是不可能避免的。人都会犯错误，你怎么能保证不出现误操作呢。看上去，他说得蛮有道理。但是，如果把你的钱都存在你自己维护的系统之上，你说你还会误操作吗？说来说去，真相只有一个——没有触及自身的根本利益而已。当你把手放到键盘上的时候，就得打起十二分的精神，一定要慎之又慎。不论何时，敬畏心都不能丢。

#### 2. 对业务需求要存谨慎心

“不得罪人的 SE，不是一个好 SE。”

运维工作经常会面对很多业务需求。当需求出现时，SE 首先要对需求进行风险分析，并对实施的合理性做出评估。绝对不是急急忙忙地就去执行任务。对于不合理的需求，我们应当提出改善意见。对于触碰底线的提案，我们应当勇于说 NO。



就拿文件传输来说，我们是明令禁止 FTP 的。但有些同事不理解、不支持我们的工作，用业务需求去打压，说代码不支持 SFTP、不支持公钥什么的。如果这些“坏了规矩”的申请被通过执行了，就是 SE 的严重失职。

现今的移动支付如此火热，越来越多的金融资产正向着互联网行业涌入。随着生物信息识别技术的发展，我们已经迈进了“刷脸时代”。在享受便捷支付的同时，个人隐私的安全也正承受着前所未有的极大威胁。手机、照片、身份证、指纹、虹膜、面部图像，我们上传的资料还少吗？一旦信息泄露，这个世界转而对将狠狠地惩罚那些已经完成了各种资料录入的用户。

每十个需求中，总有那么一两个需求是假的。在所谓代码不支持、场景不适应或者其他各种理由的背后，往往隐藏着更深层次的顾虑和担忧。其实 SFTP 公钥访问并没有什么技术门槛。只是出于对代码实现的不熟悉，或者是嫌麻烦不愿去改，进而提出了一个非要用 FTP 的“假需求”。

运维对于业务来说是职能部门，做服务工作讲究尽力满足用户需求。遇到类似的这种问题，我们要从背后去挖掘他们的真正意图。你不知道如何实现，我可以提供示例代码。你要协调第三方，我可以提供技术和法理层面上的支持。但是对于触碰底线的需求我们绝对不能低头。

SE 永远不能忘记自己的使命。你处在基础架构的最底层，也是整个平台最后一道防线，你的每一个操作和决定都对业务全局产生着重大影响。谨慎之心比敬畏之心更为重要。

### 3. 对功名利禄要抱平常心

最为关键的，我们还要抱有一颗平常心。所谓的平常心，是指你要能耐得住寂寞，经得起诱惑。想要做好 SE，你首先就得学会吃亏。一来 SE 远离业务，不是那种显山露水的岗位。再者，这一行原本就是宽进严出的。想入门很容易，但要想做出成绩来是非常难的。古人云，十年磨一剑，霜刃未曾试。十年之间若有小成，已属十分不易。所以，那些急功近利、心浮气躁之人最好就不要进 SE 的门。

## 16.1.3 匠人精神

做技术，必须要有匠人的精神，这一点对 SE 来讲尤为重要。什么是匠人精神？那是一种无论外界环境如何变化，都能坚守做事原则的骨气。

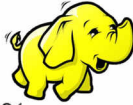
### 1. 精益求精

“只有优秀的 SE，没有合格的 SE。”

精益求精是匠人的基本素质。我中学时代的一位物理老师曾经说过，凡是期望自己能考 60 分的学生，最后大多及不了格。这句话我铭记了二十多年。但凡要做事，就要尽己所







能做到最好。但一些人却将它给歪曲了，认为如果做不到完美，我就不能交付。这句话反而成了他们拖延工期、效率低下的极好借口。我们是做工程的，不是艺术家，对生产效率是有要求的。所谓精益求精，是指做事追求极致，尽全力而为之。如果明明可以做好却没有尽力，这是我不能容忍的。

在一次硬件选型的过程中，我通过对比两款不同型号的 CPU 后发现，选择处理器 A 能够同时降低能耗和采购价格，但它的主频要比 B 低一些。而 B 的主频和我们现有处理器的主频是一致的。为了验证 A 的可行性，我特意去做了相关的压力测试。其实，两款 CPU 的差价也就是一千多元。可能有些人觉得，既然价格差不多，又何必给自己找麻烦呢？但是我不这么想。我们每年采购服务器的基数是很大的，这个差价累积起来少说也要几百万元。当然，决定权不在我，你非要用贵的，我也可以妥协。但这种观点我无法认同，因为我良心上过不去。

## 2. 严谨求真

在日常工作中，我们经常能听到这样模棱两可的回答：这个错误可能是某某原因引起的，那个故障大概是这个样子的吧。有些同学遇到问题时，缺乏严谨求真的态度。喜欢从网上搜答案，未加论证就盲目地轻信他人的结论，知其然不知其所以然。这种工作态度不是一个 SE 应有的作风。

我建议大家有时间多读读官方文档。使用搜索引擎处理故障确实方便，但是网上的东西不见得都是对的。生产环境的差异、对技术理解的偏差、对问题描述的准确性，都会对解决方案产生不利影响。网上的答案我们可以作为参考，但一定要经过验证才能实施，万万不可直接拿过来就在生产环境上运行。

遇到问题一定要寻根问底、弄个明白。“差不多”等于“差很多”。做技术的人，严谨求真的精神要永存。

## 3. 坚持专注

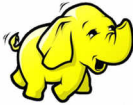
要能够隔离外界的干扰，坚持专注完成一件事情。从 0 到 1 是非常难的，但只要突破了 this 1，再从 1 到 100、甚至 1000 就很容易了。通往成功的道路肯定不是一帆风顺的，当阻力到达最大时，大家比拼的不再是技术，而是看谁能咬牙挺过最后一刻。但如果你的功利心太强，过于在意外界的评论和声音，做事的品质就下降了。

## 4. 谦逊务实

不骄傲自满、自恃才高，不断地学习新的知识，虚心向他人求教，是匠人谦逊的品德。脚踏实地，实事求是，不浮夸，不吹嘘，淡泊名利和虚无的光环，是匠人务实的态度。

一流的工匠，心性比技术更重要。好的技术，要依托在好的心性之上，否则做出的产品也毫无艺术和价值可言。只有在内心当中注入了真正的匠人魂魄，才能成为值得人们尊敬的大师。





## 16.2 未来时代

人脸识别、机器学习、神经网络，技术的风向标已经开始转向。想想还真是可怕，昨天大数据和云计算还叱咤风云，转眼间它们似乎已成明日黄花。我深深地感受到了这种飞速变化所带来的冲击。传统经济正朝着互联网的方向转型，过去的知识结构是陈旧的，对新技术、新变化的不适应，让我们感到无所适从。看身后，我们正在被无数的新人所追赶，自身的优势正在慢慢消亡。

### 16.2.1 前方高能——出现怪兽 AlphaGo

2017 年 10 月 19 日，Google 公司的 DeepMind 团队在 Nature 上发文，宣称新一代的人工智能 AlphaGo Zero，在自学了三天的围棋之后，以 100:0 的战绩完胜之前击败李世石的 AlphaGo。这则消息的推出犹如一颗深水炸弹，让很多人坐不住了。当深蓝击败卡斯帕罗夫、AlphaGo 战胜李世石和柯洁的时候，似乎我们还能够镇定自若。不过是输了一盘棋而已，比速度原本就不是我们的优势所在，再说机器也是基于人类的棋谱才赢的。如今人家 AlphaGo Zero 是自学成才，真正实现了传说中的“72 小时从入门到精通”，完全不依赖于任何已知的样本，在跟自己左右互搏了 490 万局棋之后，突然大彻大悟、能力觉醒，终成一代围棋宗师。

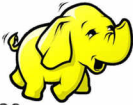
此前，霍金、盖茨和马斯克都不同程度地表示出了对人工智能的担忧。这种看法是无道理的。从第一次工业革命开始，机械化生产取代人工劳作的进程从来就没有停止过。我们一直在享受这种革命成果所带来的红利，但我们今天终于开始担心，这一次人工智能会不会革掉了人类自己的命。在没有样本束缚的情况下，人工智能特立独行，找到了目前为止围棋的最优解。也许以后，我们就没法在一起愉快地下棋了，因为你根本没机会赢。我不懂围棋，但如果 AlphaGo 跨界去干运维，可能要担心的就不是李世石们了。人类之所以恐惧，是因为我们长期局限于自己千百年以来的固有经验之中，放慢甚至停止了思考和探索的步伐。缺乏思考和学习，可能是我们最终被人工智能所毁灭的主因。

我没有定义未来的能力，但从发展趋势上看，人工智能将给我们带来一些新的变化。

#### 1. 人工智能会毁灭人类吗？

人工智能是否会毁灭人类还犹未可知，但一定会先毁灭那些无能之辈。

曾经有一个团队的领导和我说，他手下招的那几个人，虽然没什么能力，但他们听话啊，让干啥就干啥。我们一直在宣扬“忠诚大于能力”，这句话倒是没什么毛病，不过“忠诚大于能力”不等于“可以完全没有能力”。埋头傻干、勤劳苦干在昨天是优点，在今天是累赘，到了明天将变成一堆垃圾。不思考，不学习，只能机械地做出应激反应，无异于行尸走肉。某些人做工作不动脑筋，遇到异常情况，既不会变通，也不愿意向别人请教，继续按照原先的方案执行，出了问题反倒责怪别人当初没有告诉他。像这样的人，不要说人



工智能，就是写一个普通的程序都比其胜出百倍。

我曾经面试过一个候选人，他从事硬件测试六七年了，目前在带领一个测试团队。他对自己的薪水不太满意，但又拒绝学习测试领域以外的其他知识。正是因为这个原因，他最终没能通过我们的考核。

我们给出的落选理由如下：硬件测试虽然很重要，但是从我们每年在这上面投入的时间成本上估算，设立一个专职岗位是不合算的。一个薪水不低的人，却不能在“农闲时间”提供其他技术输出，这种赔本儿的买卖谁都不会做。不管你在这个领域有多牛，故步自封很快就会使你的优势荡然无存。

传统相声《连升三级》里面有个人物叫张好古，此人大字不识，却靠关系在科举考试中考中了第二名，入了国家的翰林院，而且一混就是一年多。其人子孙延绵不绝，在如今的职场里面，也有不少这样尸位素餐之人。这种人仗着一张伶俐口，混得是如鱼得水，不但不干活，还要把干活的人通通都干掉。当人工智能时代来临之时，这种人的下场将是死无葬身之地。

## 2. 人工智能会消灭程序员和运维团队吗？

程序员不会失业，但码农会。未来的程序员只需要研究一种源码，那就是人工智能的核心。运维团队同样也不会消失，但只能留下极少数的人。我在《运维前线》一书中提过，没有人工智能，自动化运维本身就是一个伪命题。真正意义上的自动化是完全不需要人为干预的。当人工智能实现了自动化运维的时候，运维的属性和内容将发生巨变。到那时，各种系统故障都会被自动处理，我们唯一需要运维的就是——人工智能不要造反才好。

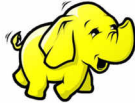
## 3. 人工智能会解放我们吗？

有人说，人工智能将解放人类，我们的工作时间将大幅缩减。这句话也许是真的，但那也是对有工作的人说的。别以为没有工作，机器人就会养你。即便你有最低生活保障，可那种日子好过吗？我们有两个地方不能和机器比，一个是比记忆容量，另一个是比计算速度。所以如果你还抱着传统观念不放，读死书、死读书，恐怕将来真的会找不到工作。

低头做事很好，但要不时地抬头看路。新时代需要的是体验和思考。比如我们有些产品设计，在宣讲的时候，售前在上面滔滔不绝、两眼放光，可用户在下面却听得哈欠连天。如果设计师没有参与过一线工作，对业务需求一无所知，也不清楚业务流程中的痛点在哪里，设计出来的产品问题多多，也就没有什么好奇怪的了。

那么光有体验就够了吗？总有那么一些人，做起事来各种抱怨，说每天都是重复性工作，既辛苦又无趣。一边干一边骂，可骂完了又接茬儿继续干。对于这种人的境遇，我只能说活该。你说他体验了么？他确实体验了，也知道痛点在哪里。但他缺乏思考，不去做任何改变。想要改变境遇，就不能等着被人救赎，必须打破惯性的思维模式。重复不可怕，可怕的是你已经习惯了。





## 16.2.2 从现在开始就要改变自己

你不改变自己，就要被别人改变。如果你觉得难受了，觉得疼了，哭闹是没用的，请当下立即做出改变。曾经的我也有过类似的困惑，所以作为过来人，我想给大家一些好的建议。

### 1. 放大格局

首先要放大你的格局，站得高才能看得远。过去我们只关注一个点，只关心具体的细节，没能从整体、宏观的角度来思考问题，没有将具体问题抽象化，把解决一件事提炼升华成解决一类事。这就是格局的差距，这个差距越大，你的困惑就越多。

### 2. 无谓加班是无能的表现

做运维工作，偶尔加班可以理解，这是不可避免的，但是加班常态化就不正常了。没有特殊情况，八小时内能完成的工作，为什么还要加班做？是工作能力低，还是任务分配不合理？在学生时代，每逢考试的时候，大家比着看谁能第一个交卷（当然不是交白卷）。怎么到了职场，加班倒成了好员工的代名词了呢？如果成功是靠加班加出来的，那你不如就住在公司里好了。用加班时间去考核员工绩效、衡量员工价值观的行为是可耻的。这不能让大家更敬业，只会降低优秀员工的工作效率，并给予那些能力差的人更多的宽容和借口。

时间是你的资源和财富，花掉的时间都是成本投入。如果你投入的成本不能换取更多的利润，那为什么还要加班呢？凡是那些写作业总写到深夜的孩子，成绩肯定都不会太好。你想，一个班 30 多个学生，如果别人都能做完，为什么你就非要拖到半夜呢？不是你的功课不扎实，就是缺乏专注力。从工作效率上就能反映出一个人的专业力和专注力的水平高低。

### 3. 自找苦吃

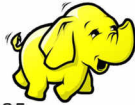
都说干运维的人工作压力大，这句话所言不虚。没有压力、整天笑咪咪的人不可能是真正的运维人。至少他绝不是一个身处在运维前线的人。

我以前去阿里面试的时候，HR 问过我一个问题——你工作中有没有压力？当时我没太理解她的意图，觉得这个问题很奇怪。如果现在这份工作我都有压力，哪有底气来参加你的面试呢？现在，我多少有点儿明白了。在这位面试官看来，如果你没有压力，说明你的工作还不到位，还没有达到全力以赴的满负载状态。欲取得更大的成就，就要逼迫自己离开舒适区。

### 4. 为自己打工

我看过一个特种部队的片子，里面叙述了这样一个情节。特种行动小组即将接受军区长官的检阅，战士们日夜刻苦训练，期望能向首长展示出他们最好的风采。可等到检阅那天，将军只看了两个人的表演。有些战士觉得很失望，而将军却说了一句震撼人心的话：





“你们刻苦训练是为了我吗？你们每天流汗流血不是为了给谁看，更不是为了取悦谁。不要忘了你们的使命，你们所做的这一切都是为了保卫国家和人民的安全。”

做事有四种态度。

- ☐ 做牛马——把工作当负担，糊弄着做。
- ☐ 做任务——把工作当差事，尽职做。
- ☐ 做产品——把工作当作市场价值，用心去做。
- ☐ 做品牌——把工作当成文化遗产，注魂去做。

有些人觉得自己在公司里干得不爽，就想出去创业。可是如果你连公司里的工作都做不好，出去创业不是更糟糕吗？其实创业无时不在，无处不在。消除为他人打工的心态，你就是在为实现自己的价值而工作。

### 5. 受尊重的都是有实力的人

我在 51CTO 高招微课上分享的时候，有个同学问说自己刚刚毕业，现在有些单位瞧不起应届生，就觉得你啥也不行，工资还低得出奇。怎么办？我给他打了个比方。我问他，假如你走在路上，看见一个小石子，是不是特想上去当球儿踢一脚？但如果换成一块儿砖，你还踢吗？毕竟你是白纸一张，别人看你不爽，把你一脚踢开也是正常的。但如果你发奋图强，终有一天成长为一块巨石的时候，尽管他们依旧可以让你滚蛋，但这次他们得学会客气一点，要用手搬、用车拉，而不能用踢的。

### 6. 我们不可能让所有人都满意

人在江湖，相互竞争，在职场里难免会遇到一些讨厌的人。你刚做出点儿成绩，他就处处打压、排挤你。不管你做什么，他都不认同。请不要太在意，做好自己的事，不要被外界的一些杂音所干扰。梁启超曾经说过：“天下唯庸人无咎无誉。”不管你多大的腕儿，有叫好儿的，就有骂街的。装睡的人是永远也叫不醒的，但是太阳不会因为你闭上眼睛就失去光芒了。

网上有两句话说得好：爱我的人谢谢你，骂我的人没关系。我们不可能让所有人都满意，因为不是所有的“人”都是“人”。

## 16.2.3 开启你的管理模式

曾经有一个小伙子来参加我的面试。当时我问他一个问题：如果你加入团队，愿意做多久？有什么职业规划么？他满怀激情地告诉我，要在一年内成为团队的核心，五年内成为架构师，第十年成为技术总监。我继续问道：团队核心需要掌握哪些技能？你打算如何实现？此时，他沉默了。从他脸上不自信的表情可以看得出，他对自己的目标定位和实现方法根本就不了解。如果今天的我作为候选人去参加面试，作为面试官，我可能会问自己这样一个问题：如果你在这里没有做到总监，你该怎么办呢？

觊觎金字塔尖的人何止千万？你要从千军万马中脱颖而出，去抢夺一个晋升名额。但不管你怎么努力，好位置早就没有了。好像除了升职和跳槽，我们已经没有其他增加财富的手段了。我问了自己一个可怕的问题，既然你无法通过工作致富，那么你的工作意义何在？如果每天循规蹈矩的上下班，穷尽一生只是为了填饱肚皮，一眼就能望穿三十年后自己退休时的样子，在我看来是可悲的。难道说我的价值就只能由别人来决定么？

做技术的人容易犯偏科的错误，在专业上下了很多工夫，却忽视了其他方面的修为。习惯于埋头做事，却很少抬头望天。有一篇文章叫《程序员如何用技巧变现》，有很多值得我们反思的地方。像盖茨、乔布斯这些人，他们的成功起源于技术，但绝不仅仅局限于此。如果他们的眼睛只盯着技术，这个行业只不过又多了几个码农而已。

### 1. 为什么你需要管理

当你被人质疑、被人否定，自觉回天乏术，对未来感到迷茫的时候。你可曾想到，你所缺乏的正是对自我的认知和管理。从小到大，可能你一直都没有好好地认识过自己。因为你懒于打理自己的人生，所以你的生活总是处于昏昏沉沉的状态。连垃圾在分类整理后都能提炼出有价值的东西来。你作为一个人，为什么不能？

一个人就是一个企业，一份工作就是一份创业。你做了这么多年的人，干了这么多年的工作，从没有想过如何实现自我价值的提升和兑现，却把希望都寄托在自己领导的身上，真是可怜可悲可叹！

---

你原本有一个远大的抱负，却终日被琐事所累，那是因为你不懂得目标管理。

每天累得像条狗，那是因为你不懂得时间管理。

工作了很多年却没什么积蓄，那是因为你不懂得财富管理。

你一直在职场上打转，不断地跳槽，工资却没涨多少，那是因为你不懂得职业管理。

你升职无望、被迫失业，顿觉天都塌了，那是因为你不懂得危机管理。

---

自我价值的实现首先要从自我管理开始，要学会管理自己的目标、时间、情绪、职业、财富等一系列的东西。

### 2. 你必须要懂管理

也许有人会说，我又不是领导，为什么要学管理？可能有些领导会说，我认为某某不适合做管理。但我要说，人人都需要、也都有权力做管理。这里不存在什么合不合适、有没有资格的问题，因为你必须懂管理。管理的对象是人，既然你是人，你交往的对象也是人，凭什么说自己可以不懂管理、不需要管理？凭什么要把管理权拱手让与他人？

管理并不是领导们的特权，它是一项人人必备的生存技能。你不但能够管理自己，还可以去管理他人。让牛人与你合作，取得更大的成绩。通过管理，你将掌控自己的命运，朝着你自己的人生道路去前行。



### 3. 不懂管理你将永远被动

你有没有想过这些问题？

---

你关注的是细节，他关注的是格局。

你研究的是技术，他研究的是市场。

你在看架构融合，他在想风险融资。

你加班加点，为刚刚解决了一个 bug 而松了一口气。与此同时的他，在饭桌前与客户觥筹交错，畅谈产品构想和行业价值。

他每天井井有条，可以调度各种资源。而你每天焦头烂额，到处看别人的眼色行事。

他不断地出入于各种高端社交场合，获取了大量的人脉资源。而你每天下班到家已是深夜，连晚饭还没有吃。

他的时间越来越充裕，有很多进修、学习、思考的机会。而你却发现，自己活得越来越累，好像身体已被掏空。

他每天上班是开心的，憧憬着未来的幸福。而你每天上班是痛苦的，想着昨天还有两个工单没有干完。

---

细思极恐啊，朋友们。当然，这还不是最可怕的。最可怕的是，你放弃了自己的管理权，而他却在研究如何管理你。你不懂管理，就只能成为一个被管理的角色。

## 16.3 写在最后的话

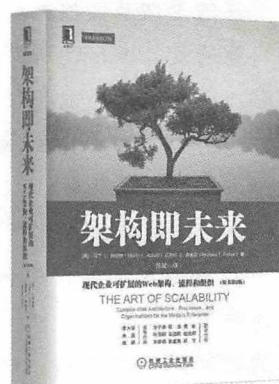
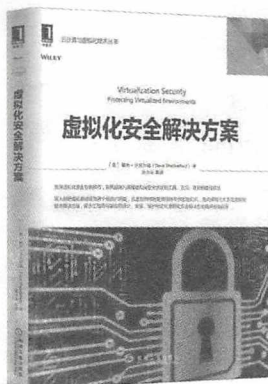
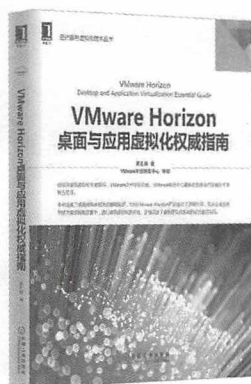
故事讲完了，我也该和大家说再见了。其实，安排这样一章内容，也有着我特别的用意。在《运维前线》一书的附录中，有一篇《求职者与面试官》，那正是在下的拙作。我很高兴看到有书评说，这篇文章写得不错。作为一篇管理类的文章，能够选入技术书籍并被读者认可，我觉得这是一个好的契机。技术这条路我会走下去，但我要考虑的是，如何才能走得更远。

新的一年，我将构思一些新的内容。我的计划是为所有的职场人，尤其是广大的基层员工和职场新人，写一部属于我们自己的管理书籍。我也期待在不久的将来，我的新书能够和大家见面。感谢你的陪伴，我们有缘再见。





## 推荐阅读



### VMware Horizon桌面与应用虚拟化权威指南

作者：吴孔辉 著 ISBN：978-7-111-51202-8 定价：59.00元

由资深桌面虚拟化专家撰写，VMware大中华区总裁、VMware研发中心高级总监等业内领袖及专家联合推荐。本书涵盖了桌面虚拟化相关的基础知识，也对VMware Horizon产品进行了详细介绍，并从企业业务与技术需求的角度着手，进行桌面虚拟化的评估，全面讲述了桌面虚拟化系统的设计最佳实践。

### 虚拟化安全解决方案

作者：[美]戴夫·沙克尔福 著 张小云 等译 ISBN：978-7-111-52231-7 定价：69.00元

资深虚拟化安全专家撰写，系统且深入阐释虚拟化安全涉及的工具、方法、原则和最佳实践。深入剖析虚拟基础设施各个层面的问题，从虚拟网络到管理程序平台和虚拟机，重点阐释三大主流虚拟化技术解决方案，能为工程师与架构师设计、安装、维护和优化虚拟化安全解决方案提供有效指导。

### 架构即未来：现代企业可扩展的Web架构、流程和组织(原书第2版)

作者：[美]马丁L.阿伯特 等著 陈斌 译 ISBN：978-7-111-53264-4 定价：99.00元

本书深入浅出地介绍了大型互联网平台的技术架构，并从多个角度详尽地分析了互联网企业的架构理论和实践，是架构师和CTO不可多得的实战手册。

——唐彬，易宝支付CEO及联合创始人

本书基于两位作者长期的观察和实践，深入讨论了人员能力、组织形态、流程和软件系统架构对业务扩展性的影响，并提出了组织与架构转型的参考模型和路线图。

——赵先明，中兴通讯股份有限公司CTO

## 内容介绍

资深系统运维专家撰写，知名运维专家联袂推荐，注重方法和思路，将枯燥的操作上升到设计和建模高度。本书站在 IT 基础架构视角，分析数据中心选型与规划、管理流程设计与实施、基础服务构建、系统运维实用经验、职业发展探讨等，大致可划分为六部分，16 章。

**第 1 章**，谈谈笔者心中的 IT 基础架构标准、写作初衷和本书特点等。

**数据中心篇（第 2 ~ 5 章）**，综合介绍数据中心、网络、系统等多个技术领域的话题。笔者曾亲手规划、建设多个同城数据中心，经验丰富，难有雷同之作。

**管理流程篇（第 6 章）**，管理流程是基础架构中重要的核心组件，剖析 CMDB、Workflow 的设计原则与注意事项，简洁而不简单。

**基础服务篇（第 7 ~ 11 章）**，基于多机房和海量节点，分享如何去构建 DNS、NTP、文件共享、配置管理等一整套服务的方法，事半功倍。

**系统运维篇（第 12 ~ 15 章）**，与日常运维管理的工作相关，诸如硬件故障处理与维修、安全、性能校准、Shell 编程等，临危不乱。

**第 16 章**，聊聊系统运维工程师应该具备的素养，如何提升自己。

本书穿插 13 个有趣的运维小故事，读累了在这里稍作安歇，里面蕴藏着很多收获呢。

## 刘浩

360 云事业部总经理

本书以提纲挈领式的全面讲解，呈现了基础运维工作的内容，并将各个要点有机串连起来，深入浅出地贡献给读者，既可以作为基础运维工作的入门书籍，又可以作为日常运维工作的参考比照。除了享受作者的文笔之外，不禁还要感谢作者为基础运维工作做出的贡献。推荐！

## 王津银

优维科技 CEO

本书作者从数据中心、管理流程、基础服务、系统运维等多个维度来讲解对运维的理解，兼顾了流程、人和技术三个要素，更以层次化的方式自底向上剖析运维，这对一个技术运维人来说是何等重要。喜欢运维的你，一定不要错过本书的精彩内容！

## 肖力

云技术社区创始人

基础设施涉及从硬件到软件、从技术到流程各个层面的知识，内容非常广泛，如何管理好基础设施，知识点非常多。本书系统化地总结了基础架构知识，相信读者阅读后能够快速成为基础架构领域的运维专家！

## 智锦

杭州云霁科技 CEO、  
资深运维从业者

本书激发了我强烈的共鸣，每一个章节都能让我回想起成长的一段经历。本书来源于一线实战，又兼具理论高度。云计算时代，对系统工程师的需求和要求越来越高，希望本书的问世可以惠及更多有志于从事云计算运维的同行，传道授业解惑，让天下没有难运维的数据中心！

投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)



上架指导: 计算机/云计算

ISBN 978-7-111-59778-0



9 787111 597780

定价: 99.00元